Web Development

Master's degree in Mobile Computing – Computer Science

Guilherme Ferreira Santo

Leiria, september of 2023

Web Development

Master's degree in Mobile Computing – Computer Science

Guilherme Ferreira Santo

Internship Report under the supervision of Professor Luís Frazão from Polytechnic Institute of Leiria and master Gonçalo Dias from xgeeks

Leiria, september of 2023

Originality and Copyright

This internship report is original, made only for this purpose, and all authors whose studies and publications were used to complete it are duly acknowledged.

Acknowledgments

I would like to give a special thanks to my supervisor Luís Frazão, my engineering manager Gonçalo Dias, and the colleagues I worked with on the various projects.

Abstract

In this thesis, I describe my journey during a nine-month internship at xgeeks. The dynamic nature of modern software development has elevated the role of a Fullstack web developer to a pivotal position. As I embarked on this internship, I faced the challenge of transitioning from a newcomer to a proficient web developer. Motivated by the realization of web development's critical role in modern businesses, I eagerly embraced the opportunity.

The internship took place in an environment where the role of a Fullstack developer was central, acting as a bridge between frontend and backend expertise, crucial for the seamless functioning of modern applications. By having good communication skills as one of my strengths, it allowed me to integrate seamlessly into their work ethic and diverse development teams. Throughout the internship, I contributed to three distinct projects. The first was SPLIT, an open source retrospective tool, where I was a Fullstack developer. The second project was Command Center, an application that served as a hub for various internal applications, where I worked as a Fullstack developer, but with more focus on the frontend of the application. The third and final project was a website for xgeeks' yearly event called Geekathon, where I worked solely in the frontend of the application. All projects followed Agile development frameworks, such as Scrum, Kanban and Scrumban. In hindsight, the internship was a success because all objectives were accomplished.

I transitioned from a novice to a proficient web developer by acquiring proficiency in a range of technologies and frameworks such as HTML, CSS, JavaScript, TypeScript, React, Next.js, and NestJS. The application of Agile methodologies in real-world development scenarios provided a tangible realization of my theoretical knowledge. Across all projects, I actively participated in idea generation, wrote clean and maintainable code, resolved bugs, and maintained effective communication within the teams.

My internship experience at xgeeks not only solidified my career aspirations but also emphasized the significance of adaptability, continuous learning, and effective communication in the realm of modern software development.

Keywords: Internship, Web Development, Fullstack, Frontend, Agile

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations and Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Delivery
CI	Continuous Integration
CLI	Command Line Interface
CSS	Cascading Style Sheets
CSR	Client-Side Rendering
DOM	Document Object Model
DTO	Data Transfer Object
FAQ	Frequently Asked Questions
GIS	Geographic Information System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSX	JavaScript XML
OOP	Object-Oriented Programming
PHP	Hypertext Preprocessor
PR	Pull Request
RAM	Random Access Memory
REST	Representational State Transfer
SDK	Software Development Kit
SEO	Search Engine Optimization
S3	Simple Storage Service
SSO	Single Sign-On
SSR	Server-Side Rendered
TSX	TypeScript XML
UI	User Interface
URL	Uniform Resource Locators
WAI-ARIA	Web Accessibility Initiative – Accessible Rich Internet Applications

XLM	Extensible Markup Language
XP	Extreme Programming

1. Introduction

In the dynamic landscape of modern software development, the role of a fullstack web developer has emerged as indispensable in order to create robust and user-centric web applications. Within the context of xgeeks, the company where I did this nine-month internship, the role of a Fullstack developer held a central place within the structure of the company, standing as the second most common role, behind backend developers. In today's tech-driven world, where user expectations are sky-high and the digital ecosystem is continually evolving, the role of a fullstack developer has evolved into a critical nexus between frontend and backend expertise. With that said, Fullstack developers are pivotal contributors to a company's technological advancement and success.

Before starting the internship, I knew it was going to be a challenge, because I didn’t have extensive experience in web development. Instead of being fearful, I was strongly motivated because I quickly learned that web development is a crucial role for a lot of companies. Besides that, it was also my first time working in a company, so I had no clue of what to expect from this experience. Fortunately, I fit in with the company work ethic and every team I was part of. Communication is among my notable strengths, resulting in faster learning because I was never afraid of asking questions in order to continuously learn and increase my skills in this area of development.

During the nine-month internship, the goal was for me to adapt to the company’s work ethic and environment, learn web development from scratch and integrate teams that were developing existing projects. I contributed to the development of three different projects. The first two, I performed the role of a Fullstack developer, working both on the frontend and backend of the applications, and in the third project I was assigned the role of a frontend developer, having the responsibility of building the application with the exact design that was made by the designer, pixel by pixel.

The structure of the document is organized in the following chapters: the second chapter characterizes the host institution, xgeeks; the third chapter showcases the internship programme; the fourth chapter is the state of the art of every technology and framework I worked with during the internship; the fifth chapter talks about SPLIT, the first project I was assigned to during the internship; the sixth chapter describes my work at Command Center,

the second project I contributed to; the seventh chapter describes my work at the third and final project I was part of during the internship, the Geekathon website; finally, the eight chapter finishes with a conclusion to the work done.

2. Characterization of the Host Institution

The host institution of this internship was ‘xgeeks’, one of the seven core companies of KI Group. Although KI group was founded in 1999 by Dirk Buschmann, xgeeks was only founded recently, in 2019, by Miguel Loureiro. Initially, there were only 4 people in the team, but its rapid growth led to a current number of over 100 employees. Currently it has 3 offices, all in Portugal, in Viseu, Leiria and Lisbon. It is a software engineering company that focuses [1] on e-commerce and Cloud Native technologies that give businesses an edge in innovative digital experiences.

In terms of human resources, as of May 3rd, 2023, there were 110 employees working on more than 15 projects. The average age of the employees corresponded to 30 years old, and the gender balance was around 15% female and 85% male, with the goal of decreasing this gender difference as time went by. When it comes to roles, by far the most common were frontend, backend, and Fullstack. One of the main objectives for 2023 was to increase the number of DevOps due to market demand. A nicer way to look at these statistics is shown in Figure 1, which was taken from the company’s presentation PowerPoint that was given to me during my stay at xgeeks.

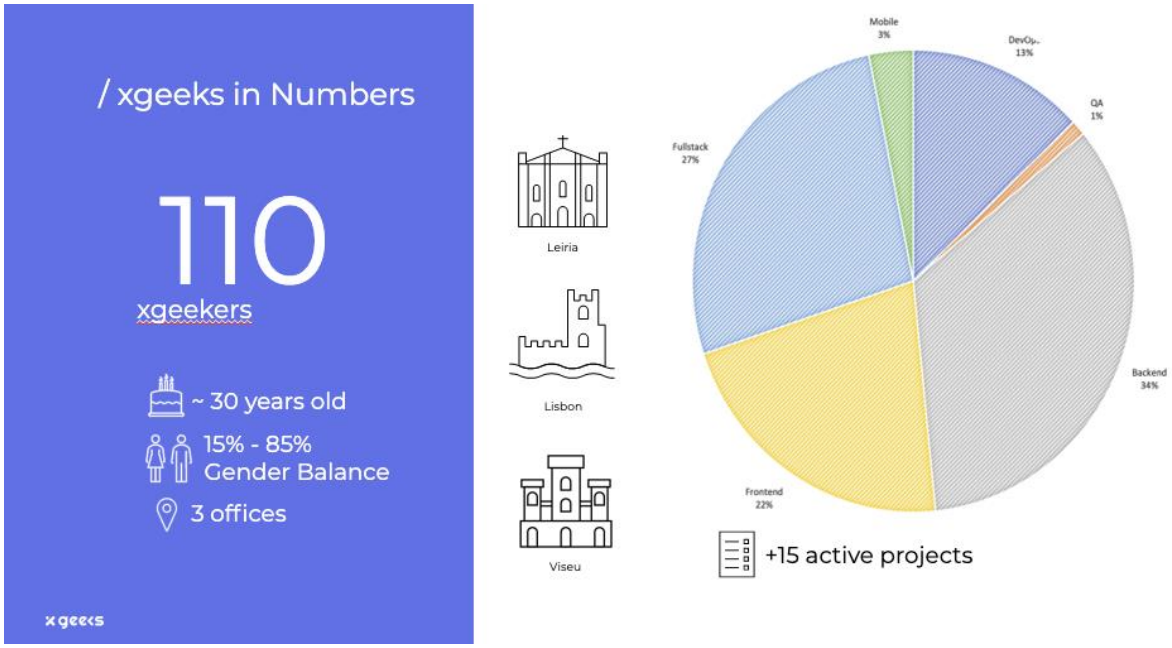

Figure 1 - xgeeks statistics (May 2023)

The company has a wide variety of clients, and by being part of a German group, the vast majority of projects came from German clients, such as:

- Daimler;
- Porsche;
- Mercedes-Benz;
- Deutsche Telekom;
- DHL;
- CheMondis;
- Cazoo.

3. Internship Programme

This chapter gives an overview of the tasks performed during the internship, which can be seen in Table 1. The number of days (excluding holidays and weekends) it took for each task is also shown in the “duration” column.

Table 1 - Internship's activities

	Start Date	End Date	Duration
Onboarding	03/10/2022	07/10/2022	5 days
Training	10//10/2022	11/11/2022	24 days
Development of “SPLIT”	14/11/2022	06/01/2023	38 days
Development of “CMD Center”	09/01/2023	05/05/2023	82 days
Development of “Geekathon”	08/5/2023	03/07/2023	40 days

3.1. Onboarding

The internship began with the onboarding phase. This is a process which xgeeks is very proud of, since they put a lot of effort into making sure every new joiner has a great integration into the company, and personally I approve that the onboarding is, in fact, a great experience. This phase, which has the duration of one week, can be summarized in the following tasks:

- Office tour;
- Lunch with the team, networking;
- A buddy being assigned to me, to give help and guidance when needed;
- Equipment and development environment setup;
- Access to tools and platforms;
- Reading documentation and policies;
- Multiple intros per day of the various projects the company was working on.

3.2.Training

Given that the purpose of my role in the company during the internship was to be a Fullstack web developer, that was exactly what my training was about – Fullstack web development. Considering the fact that I did not have major experience with web development, I had to start with the basics, and gradually learn more complex topics. The training period consisted of me studying the resources provided by the company, which were online video courses the company bought from multiple different websites. One minor hiccup to point out is that the company did not provide a specific learning path, or in other words, a recommended order to watch the courses, especially for beginners in a new role like was my case with web development. This led to me learning technologies out of order, making it a little harder in the beginning to catch up and connect all the pieces together in order to understand how everything is tied together, regarding the technologies/frameworks. Initially, there was no certainty on what project I would be part of, but there were two main possibilities, and both of which used Next.js and NestJS and their frameworks, and Typescript as programming language, so I decided to centre my learning on those 3 topics.

I started by watching an introduction course to Next.js, to learn the main characteristics of this framework, which I did, but by the end of the course I realised I had to first learn Typescript to better understand everything. With some research at the time, I learned that Typescript is an object-oriented programming language that expands on JavaScript by including optional type annotations, interfaces, classes, and other features, so I figured it was best to have a good understanding of JavaScript before diving into TypeScript. At the time, the language I had most knowledge was Java, by a long shot. As I rarely used JavaScript, I figured I should learn JavaScript all over again as a beginner, so I studied courses regarding the fundamentals and intermediate topics of it. With that out of the way, I then went to do the same thing, but for TypeScript. By the time I completed the TypeScript courses, I decided to study React courses, because as I learned in my first course (introduction to Next.js), Next.js is a React-based framework for building server-rendered and statically generated web applications, so it made complete sense to understand the fundamentals of React. After completing the React course, there was only one framework left to learn, which was NestJS. With quick research, I concluded that NestJS is a Node.js framework used for building efficient and scalable server-side applications, using TypeScript. With that information, it made sense to first learn the basics of Node.js, and when I completed it, I then watched the official NestJS course, made by the NestJS creators.

Given that I still had one week left before being integrated into a project, I decided to revisit Next.js, and do an in-depth course of it, which contained 24 hours of content.

3.3.Development

With the training phase complete, I finally started working on a project, which was SPLIT, an open-source retrospective tool for big companies. The first days were given solely to setup the project, explore the code and the application. I joined this project along with another intern, and we were both given the task of starting and finishing the user's page on the application, an objective that was accomplished. Both of us had no prior working experience in web development, so for the full duration we both did pair-programming, together.

After my job was done in SPLIT, my manager decided to move me into another internal project, but this time with a longer time horizon. This new project was called "CMD Center", and it was meant to be a hub for multiple internal applications for xgeeks. The first of those applications to be developed was "xAchievements", which I was part of, since there were no other CMD Center applications being built at the time. The use case of xAchievements was to assign achievements to the employees by tracking multiple activities they do at the company, for example, attending events, doing interviews, coding, etc. These achievements also have points associated to them, with the intent of someday creating another CMD Center application, the "xStore", where the employees will be able to redeem those points earned through achievements and collect various physical/digital rewards.

During the month of April, the company decided to stop the development of all internal projects, which lead to me jumping into the last project of the internship. This last project was the development of the Geekathon's website, which is a yearly event hosted by xgeeks where anyone can participate in a challenge to come up with an innovative idea and code it, usually in one or two days. For this project, as opposed to the other two, I was assigned the role of frontend developer.

Each project has a dedicated chapter in this report, so more details will be given regarding each project.

4. State of the Art

The purpose of this chapter is to explain all the technologies that were part of the projects I worked during the internship. It starts by giving some context regarding Web and its architecture, followed by Typescript, the programming language used in all projects. After that, subchapters for both the frontend and backend will be presented. Finally, the last subchapter explains all the methodologies used. All information is duly referenced from trusted sources of information, such as journals, reports, articles, books, and official documentation, with minor references to websites.

4.1. Web Architecture

Web Architecture can be described as a system of protocols and components that facilitates communication between client and server applications over the Internet [2]. The client and the server are the two fundamental parts of web architecture at a high level. The primary distinction between the two is that whereas a server provides access to its services by listening for connections and responding to requests, a client initiates communication by initiating a request. HTTP (Hypertext Transfer Protocol), which is a communication protocol used by web applications to transmit data over the Internet, is the primary communication protocol between the client and server. Both the request and the response have a body and headers. The content type, data size, and other characteristics about the request or response are all contained in the headers. The actual data being conveyed is in the body. The architectural design approach known as REST (Representational State Transfer) is frequently utilized for creating web APIs (Application Programming Interface). Resources, which are often available through a few clearly defined endpoints, are the foundation of RESTful APIs. These endpoints specify the actions that can be taken with respect to a specific resource, including data retrieval, updating, and deletion. APIs are frequently created to be used by other software programs, which can communicate with the API's endpoints using GET, POST, PUT, and DELETE standard HTTP methods. This makes it simple for programmers to create scalable and interoperable software programs that can speak common web protocols to interact with other systems.

4.2. Programming Languages

Throughout the internship, TypeScript was the only programming language I had to use on all projects I was part of. Because TypeScript is built on top of JavaScript, first I will give a quick overview on JavaScript before going more in depth into TypeScript, just to give some context about what TypeScript is built on top of.

JavaScript is a high-level, interpreted programming language that is primarily used to develop dynamic and interactive websites [3]. It is executed directly, line by line, without the need for compilation into a machine-readable format beforehand. It was introduced in 1995 by Brendan Eich while he was working in Netscape Communications [4]. JavaScript can be used for a wide range of tasks including server-side programming, game creation, mobile app development, and online development. All the major web browsers support it, making it the most widely used programming language in the world. With DOM (Document Object Model) manipulation and user event handling, JavaScript is largely used to provide interactivity to web pages. Moreover, it works in conjunction with HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets) to develop cutting-edge, responsive online apps.

TypeScript is an open-source and statically typed programming language, built on top of JavaScript, first released in 2012 by Microsoft [5] to enhance the development process for complex JavaScript apps. It requires type declarations and undergoes a compilation process to check for type errors before execution.

Arguably, the main feature and distinction of TypeScript is its type system [6]. This makes it simpler to identify and address problems early in the development process by catching type-related errors at the compile stage rather than the runtime. As a result, the overall quality and maintainability of the code are significantly increased. There are also more important features of TypeScript worth referencing, such as:

- Scalability - is another crucial point to reference regarding TypeScript, because it is a major advantage over JavaScript. Large applications may be written, maintained, and scaled more easily because to TypeScript's improved syntax and static type verification. The type system also contributes to the reduction of type-related mistakes that might result in bugs and other problems, enhancing the general dependability of TypeScript code;

- Support for OOP (Object-Oriented Programming) – the existence of concepts like classes, interfaces, and inheritance is another crucial aspect of TypeScript. Complex applications may be modelled and structured more easily as a result, which also makes the code easier to develop, read, and maintain. Additionally, TypeScript provides functions like generics, which let programmers create reusable code that interacts with various kinds of data, something that was often used during the development of projects in the internship;
- High compatibility with existing JavaScript tools and libraries is another plus for TypeScript. This enables using JavaScript libraries with TypeScript code and integrating TypeScript into already-existing projects simple. Moreover, TypeScript compiles to JavaScript, enabling the execution of TypeScript code on any platform that supports JavaScript.

Due to its type system and compatibility with JavaScript code that already exists, TypeScript has seen a major rise in popularity in recent years, currently being a top 5 programming language as of the yearly survey [7] carried by Stack Overflow (Figure 2), one of the world's most used website for programming questions and answers. An impressive place, considering that 4 years prior to that, in 2018, TypeScript had half the popularity [8], not even enough to be in the top 10, which can be seen in Figure 3.

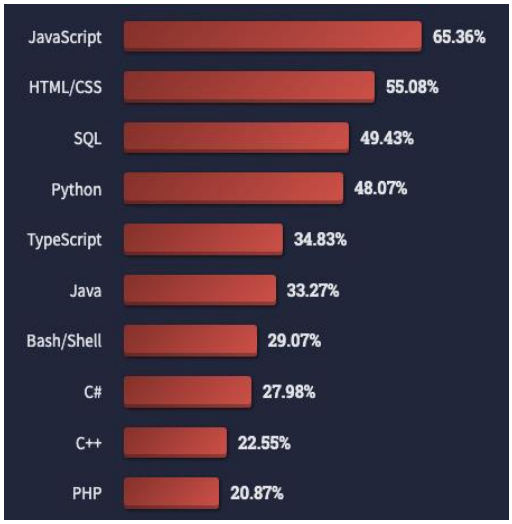

Figure 2 - 2022 most popular programming, scripting and markup languages

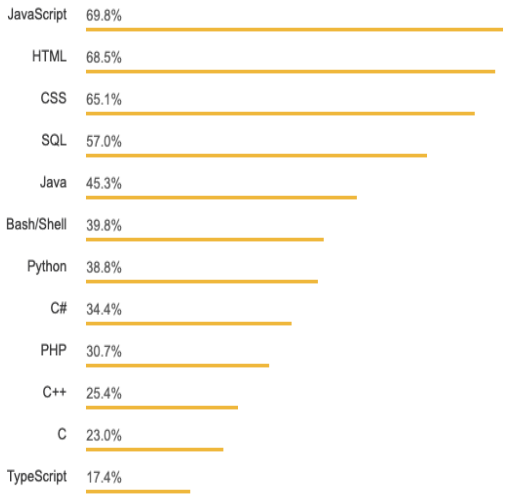

Figure 3 - 2018 most popular programming, scripting and markup languages

As seen in the above figures, Typescript is becoming increasingly more adopted as the years go by, thanks to its aptitude of catching errors early in the development process and improve the maintainability of code. In addition to interoperability with well-known frameworks and libraries like Angular, Vue, and React, TypeScript's expanding ecosystem and community support have also contributed to its increased acceptance.

Below is a comparison table between JavaScript and TypeScript, that highlights some key differences between the two. This table contains information from a particularly interesting research paper of Bogner & Merkel [9] that compares the software quality of JavaScript and TypeScript by data mining 604 open-source GitHub applications and extracting conclusion through data analysis. Besides that source, some information also comes from the popular computer science portal “GeeksforGeeks” [10].

Table 2 - JavaScript vs TypeScript

	JavaScript	TypeScript
Type system	Dynamic: types are inferred at runtime	Static: types are declared at compile time
Compatibility	All modern web browsers and server environments	Can be compiled to JavaScript and have the same compatibility as JavaScript
Performance	Generally faster than TypeScript because it has less overhead from type checking	Can take longer to compile the code due to additional overhead from type checking
Features	Functional programming, multi-paradigm, asynchronous programming, and more	Supports all JavaScript features and adds more features such as interfaces, enumerations and generics
Code understandability	Worse than TypeScript, especially in more complex code	Significantly better because types give a clear context

Learning curve	Relatively easier for beginners because it is less complex and has less features to learn than TypeScript	Requires more time and effort to learn due to the added complexity of the type system

Finally, to conclude this subsection, still related to the Bogner & Merkel paper mentioned above, I want to point out the conclusion that was made on that study. The results provide some evidence for the potential benefits of using TypeScript in terms of code quality and understandability, although they also suggest that the relationship between static typing and bug reduction may be more complex than previously assumed. In other words, when using TypeScript over JavaScript, it is not necessarily true that it will always be easier and faster to debug or prevent the number of bugs of an application, as opposed to popular belief. Albeit the study was based on a specific set of open-source projects on GitHub, the findings may not necessarily generalize to all TypeScript and JavaScript projects.

4.3.Frontend

Regarding the frontend technologies, there are a variety I used: React, Next.js, React-Query, Tailwind CSS, HTML, Radix and Stitches, but the one I used mostly is Next.js, so that will be the main focus of this subsection.

4.3.1. React

React is one of the world's most popular JavaScript libraries, being only surpassed by Node.js according to the 2022 developer survey of Stack Overflow [11], as we can see in Figure 4. It's also important to point out that in 2018 [12], as seen in Figure 5, Angular was more popular than React, but fast forward 4 years and React has more than double the popularity compared to Angular.

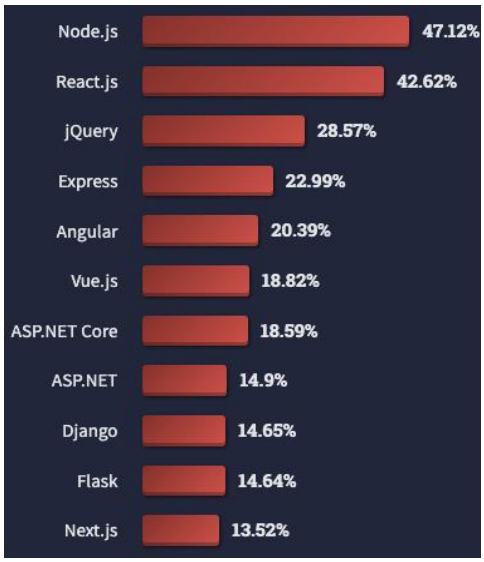

Figure 4 - 2022 most popular Frameworks, Libraries and Tools

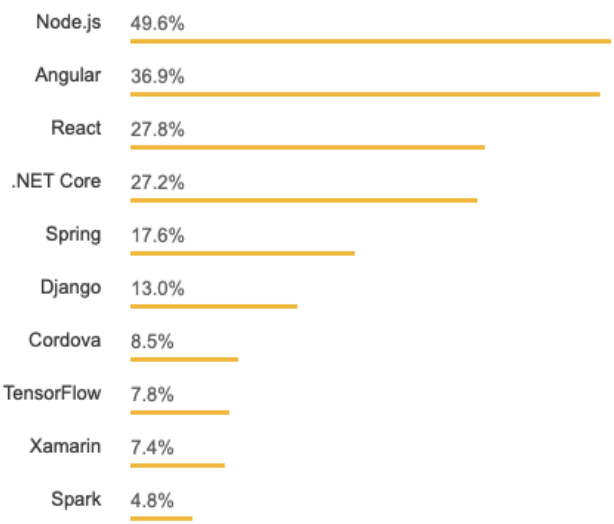

Figure 5 - 2018 most popular Frameworks, Libraries and Tools

In 2013, Facebook initially created it and made it available as open-source software. It makes complicated user interfaces easier to build and maintain due to React's declarative approach to UI (User Interface) development. React renders components effectively using a virtual DOM, enabling quick updates and little re-rendering. React's main features [13] are the following:

- Declarative - its declarative approach to constructing user interfaces, which makes it simpler to write, comprehend, and maintain complicated user interfaces, is one of its most significant characteristics;
- Virtual DOM: it enables quick and effective user interface modifications. Performance is enhanced, especially for big and sophisticated user interfaces, thanks to the virtual DOM, which reduces the amount of actual DOM manipulation required;
- JSX (JavaScript XML): it is a JavaScript syntax extension that enables the writing of HTML-like components in JavaScript code. Component definition and rendering in React are done through JSX. JavaScript functions are created from JSX components so they can be invoked to render the component;
- One-way Data Binding: in React, data flows in a single direction, from the parent component to its children, by using props. This helps in maintaining a distinct division of duties and inhibits unforeseen behaviour. When the data changes, React's one-way data binding works by re-rendering the component tree. This

procedure is automatically optimized by the React virtual DOM, which only updates the user interface elements that are impacted by the change.

Not only are the previous list's features important to know, but also the practical programming characteristics too. Some of the things that developers commonly use in React include [14]:

1. Components: React components are the building blocks of an application's UI, and can be defined as either class components or functional components;
2. Props: Props are the inputs that are passed down to a component from its parent, and can be used to customize the behaviour of the component;
3. State: State is an internal data store that is specific to a component, and can be used to manage the component's behaviour and respond to user input;
4. Lifecycle methods: React provides a set of lifecycle methods that allows to hook into different stages of a component's lifecycle, such as when it is mounted, updated, or unmounted;
5. Hooks: React Hooks permit to use state and other React features in functional components, which can simplify the code and reduce the need for class components;
6. Context: React Context enables the sharing of data between components without having to pass props down through every level of the component tree;
7. Router: React Router is a popular library for managing client-side routing in React applications. Often used to get parameters from the URL or programmatically navigate from one page to another.

Some of the most common hooks [15] that natively come with React are "useState", "useEffect", "useRef" and "useMemo". They were commonly used throughout the development of the internship projects and will be seen with some frequency in each project's section. A summary of each of those hooks can be seen as follows:

- useState: add state to functional components. It takes an initial value as a parameter and returns an array with the current state and a function to update it;

- useEffect: perform side effects in components, such as fetching data from an API or manipulating the DOM. It takes a function as a parameter and runs it after every render;

- useRef: create a reference to a DOM element or a value that persists between renders. It returns a mutable ref object that can be updated without triggering a re-render;

- useMemo: this hook allows to memoize (cache) a value, which means that it will only be re-computed when its dependencies change. It takes a function and an array of dependencies as parameters and returns a memoized version of the value.

To conclude, an important mention to tests in React, as they are a key aspect in maintaining application reliability. In order to test React applications, React Testing Library alongside other testing libraries like Jest are a common option. React Testing Library is a testing utility library [16] that enables developers to test React components by interacting with them as a user would. It provides a set of functions to render components in a virtual DOM and then simulate user interactions with those components, such as clicking buttons or typing in inputs. The library focuses on testing the behaviour of the components rather than their implementation details, making tests more reliable and less brittle to changes in the code. Jest is a popular JavaScript testing framework [17] used to test JavaScript and TypeScript code, including React projects. It provides an easy-to-use and fast test runner, assertion library, and mocking capabilities, among other features. In React projects, Jest is often used for unit testing, where individual units of the code, such as components, are tested in isolation. Jest can be used to create test suites, test cases, and assertions to ensure that the components function correctly and meet the expected requirements. Jest is commonly used alongside other testing libraries such as React Testing Library or Enzyme to perform unit testing in React projects.

4.3.2. Next.js

Next.js is a React framework [18] that provides a collection of powerful features for building static and SSR (Server-Side Rendered) applications. It's designed to enhance developer experience with features like SSR, TypeScript support, smart bundling, and route prefetching, while also introducing its own opinionated approach to application organization

and routing. Next.js provides automatic routing using a file folder system, making it easy to structure pages and fetch data on the server-side for each request. Whether building a simple website or a complex web application, Next.js offers a comprehensive set of tools and optimizations to help create highly dynamic and performant web applications.

When it comes to popularity, although React is way more popular than Next.js, as seen in Figure 4 of the previous subsection, one interesting aspect is that the worldwide Google searches between the two have increasingly more disparity, in favour of Next.js, which is shown is Figure 6. This means that although React is more used and popular than Next.js, the latter has been gaining more relative popularity.

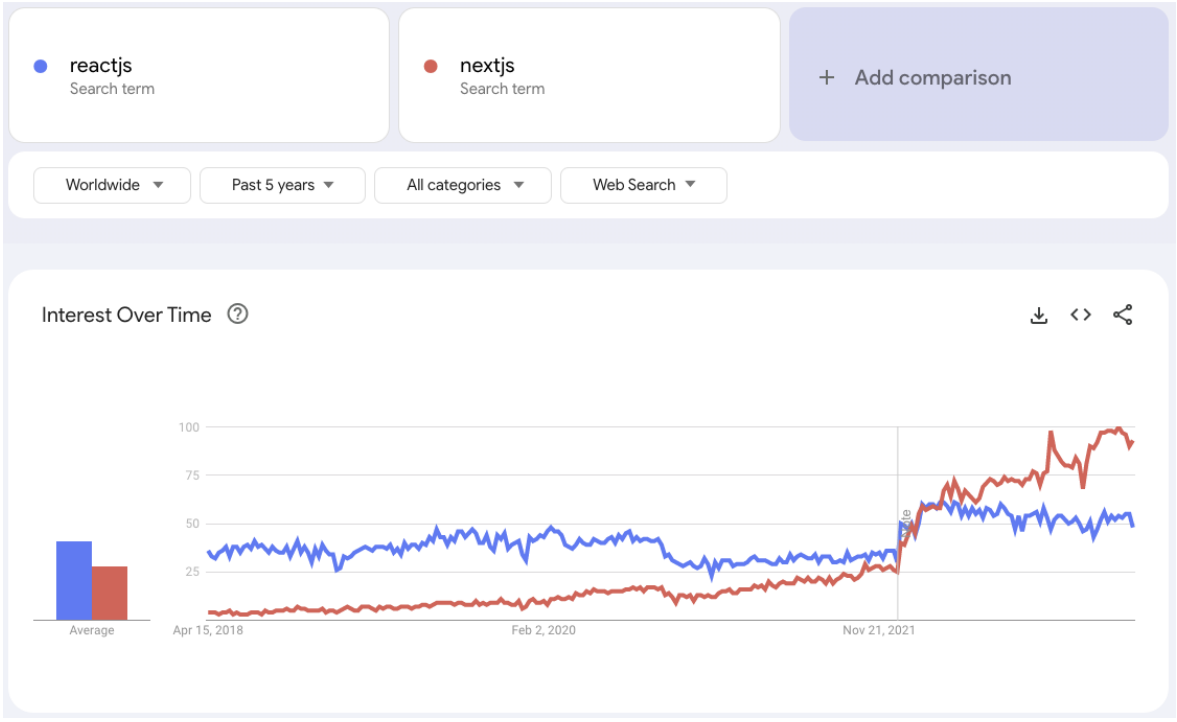

Figure 6 - Google Trends graph of ReactJS and Next.js (April 2018-2023)

Next.js has three core features that make this framework a popular choice when building React applications, which are Data Fetching, File-based routing, and API Routes. These key features will be explained after first mentioning CSR (Client-Side Rendering) and SSR, as they are a crucial element in order to understand the bigger picture of how React and Next.js applications work.

React uses client-side rendering, which runs on the browser. This technique results in a slower initial page load, but subsequent page loads are faster. The client-side framework refreshes specific DOM elements with updated data without reloading the entire user interface. The DOM is an interface that represents the HTML structure of a web page as objects that can be accessed and manipulated by scripts. In contrast, Next.js uses SSR, where the server sends the HTML response to the browser to form and render the page. The main difference between SSR and CSR [19] is that the server sends a ready-to-render HTML web page for SSR, while the browser receives a sparse document with a link to JavaScript for CSR. This allows the user to begin seeing the page while the rendering process takes place with SSR. However, with CSR, all events must take place before the virtual DOM is sent to the browser's DOM for the page to be accessible. Figure 7 and Figure 8, taken from the journal mentioned in this paragraph, are a visual representation of how CSR and SSR work, respectively.

On the same note, a key feature of Next.js is server-side Data Fetching, contrary to React which natively only offers client-side data fetching. Next.js has in addition to client-side various server-side data fetching mechanisms, giving developers flexibility. The following list explains all data fetching mechanisms Next.js provides:

1. Server-side rendering (SSR) [20]: Server-side rendering involves fetching data from the server-side to pre-render pages with the necessary data, which is useful when a web application needs to serve pages with data that must be retrieved at the time of request to the server. Next.js utilizes a special function called “getServerSideProps” to perform data fetching from the server-side. This function runs exclusively on the server-side (never runs in the browser) and can only be applied to page-level components. The “getServerSideProps” function is executed before the page is loaded, and after a delay, the data is served with the web content, making the SSR data fetching mechanism suitable for applications with a high data requirement. In contrast, client-side rendering (CSR) hits the API after the page is loaded, making it less efficient for applications that require a large amount of data.
2. Static Site Generation (SSG) [21]: Next.js also provides SSG capabilities with a special function called “getStaticProps”. SSG allows to pre-render pages at build time, fetching data from the server-side just like with SSR, but with the difference that the data is fetched only once and then served from the cache for subsequent requests. This mechanism is suitable for web applications with data that doesn't

change frequently and tends to be static, as it improves SEO (Search Engine Optimization) and application performance by rendering data on the server-side during the build process. Since the data fetching process is carried out only once during the build, it results in faster load times and better user experience. Overall, SSG is a useful tool for building performant and SEO-friendly web applications with static data requirements.

3. Incremental Static Regeneration (ISR) [22]: it is an enhancement to SSG that allows developers to update static pages without having to rebuild the entire application. This is achieved by specifying a revalidation interval using the "revalidate" prop in the “getStaticProps” function. ISR enables developers to generate dynamic content at build time while still leveraging the benefits of static site generation, such as fast load times and improved SEO. By allowing for selective and automatic updates to individual pages, ISR improves the developer experience and enables faster, more efficient updates to static content.
4. Dynamic routing [23]: it allows developers to generate dynamic pages based on user input. With dynamic routing, a single page component can be used to render multiple pages with different content based on the URL (Uniform Resource Locators) parameters. To use dynamic routing with data fetching, Next.js provides the “getStaticPaths” function, which generates a list of paths to pre-render during the build process. This allows developers to generate pages on demand based on user input, while still leveraging the benefits of static site generation, such as fast load times and improved SEO.
5. Client-side rendering (CSR) [24]: particularly useful when web pages do not require SEO indexing or pre-rendered data and when web page content is frequently updated. CSR can use React hooks' useEffect function to fetch data from APIs on the client-side, and the data is fetched every time a page request occurs from the client-side. Data fetching can be done at the component and page level, with components fetching data during installation (mounting process) and pages fetching data at runtime while the application is running, and the page content is updated as the data changes.

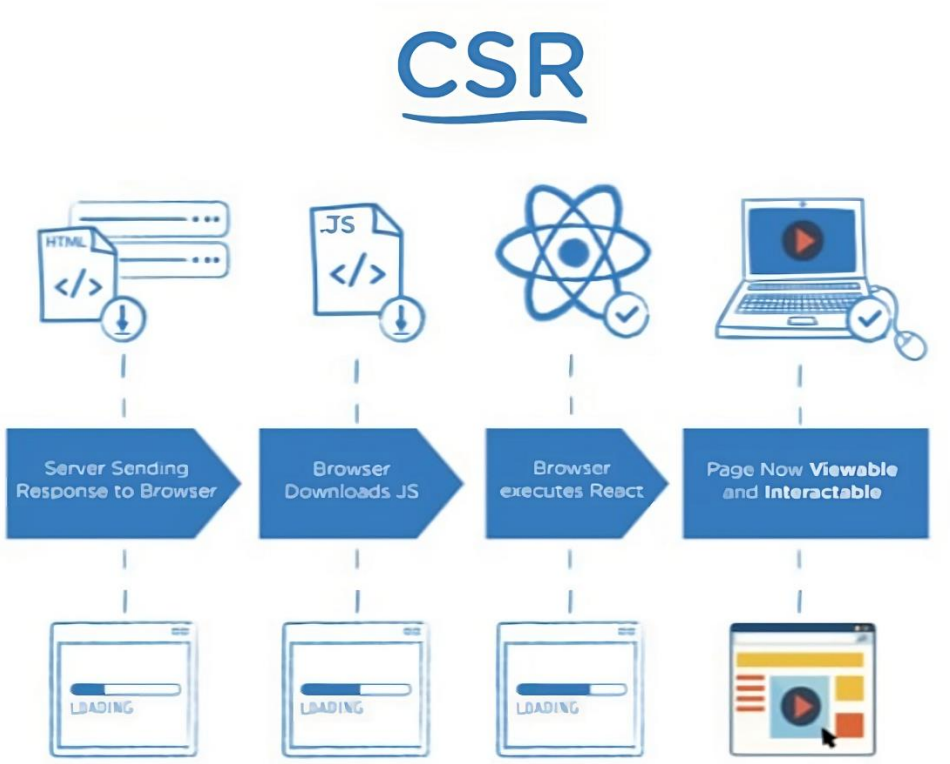

Figure 7 - Client-Side Rendering

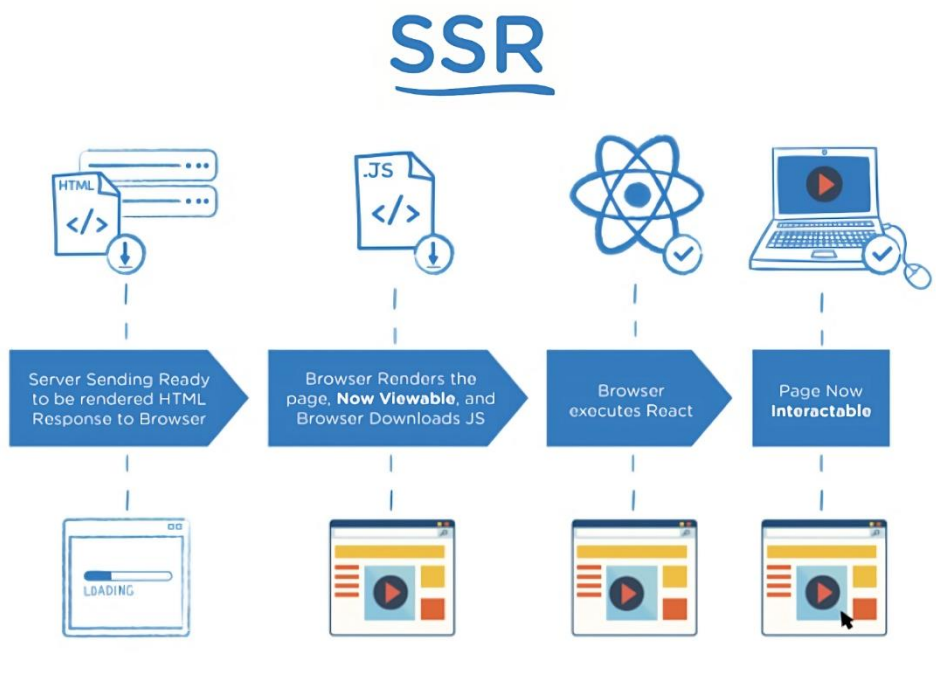

Figure 8 - Server-Side Rendering

File-based routing, provided by Next.js [25], makes application routing intuitive and removes the need for a third-party library for routing. Each page of the application is represented by a file under the “pages” directory, with the file name corresponding to the URL path. Routing in Next.js follows three important rules:

1. Index routes: files named “index.js” will automatically be routed to the root of the directory. For example, the folder structure “pages/index.js” will correspond to “/” in the URL. Another example, “pages/support/index.js” will correspond to “/support”;
2. Nested routes: files will automatically be routed the same way as a nested folder structure. For example, “pages/account/settings.js” will be “/account/settings”. Another example, “pages/dashboard/settings/username.js” will have the URL “/dashboard/settings/username”;
3. Dynamic routes: a named parameter can be used to match a dynamic route, by using the bracket syntax. For example, given “pages/users/[id].js”, a page of a user with id 1 will have the URL “/users/1”. There is also the option to catch all routes [26], which is done by adding three dots inside the brackets. For example, “pages/post/[...slug].js” will match not only to “/post/a” but also “/post/a/b”, “/post/a/b/c” and so on.

The third key feature of Next.js is called API Routes, which provide a simple and efficient way to create API endpoints [27]. This can be handy specially for relatively small projects because this way it is not necessary to create another project to serve as the backend of the application. The way API routes work is as simple as placing a file in the pages/api folder, then file is automatically mapped to the /api/* route and treated as an API endpoint instead of a page. These routes are server-side only and will not increase the size of client-side bundles. The API routes export a default function that receives the request (req) and response (res) inputs, allowing developers to control several HTTP methods within an API route. Additionally, API routes can be used to mask external URL services, providing a secure way to access external services using environment variables on the server. Overall, Next.js API routes are a powerful tool for creating robust and efficient APIs within Next.js applications.

Considering the information provided until now, as well as the journal articles [19] and [28] who both analyse Next.js advantages and disadvantages, the next two paragraphs will highlight them. Starting with the advantages:

- Lazy loading: Next.js supports dynamic imports, where components are imported during runtime or whenever necessary, improving the initial loading performance by decreasing the amount of JavaScript needed to render the page;
- Fast refresh: by default, when code changes are made to React components, they are visible on the application usually within a second;
- Built-in CSS: Next.js enables importing CSS styles from a JavaScript file, allowing for faster rendering because the browser only needs to make one HTTP request to retrieve the code;
- Image optimization: Next.js provides an Image component, which significantly reduces image size while maintaining the quality, resulting in faster page loads;
- Easy routing mechanism: Next.js has easy routing mechanism support located in the pages folder, so it doesn't need a third-party library to do routing;
- Various data fetching mechanisms: Next.js supports data fetching mechanisms by CSR, SSR, SSG, and ISR, making it highly customizable and adaptable to different application needs.
- Good for SEO: with SSR, the server sends a fully rendered HTML page to the client, which means that search engine crawlers can easily read the page's content and metadata. In contrast, with CSR, the server only sends an empty HTML page, and the client-side JavaScript code is responsible for rendering the content, which can make it more difficult for search engine crawlers to index the page's content. Additionally, SSR can improve the page load time, which is another important factor for SEO.

The disadvantages of the Next.js framework, based on the same literature mentioned in the advantages, are explained as follows:

- High server load: Using the SSR method can increase the load on the server due to frequent requests;
- Wait time for interactivity: Pre-rendering with SSR can lead to a delay in interactivity until the browser successfully executes React;

- Full page reload: When changing pages, users will experience full page reloads due to requesting a rendered HTML file from the server;
- Cost of flexibility: Next.js lacks pre-built front pages, which means building the entire front-end layer from scratch;
- Lack of built-in state manager: Next.js does not have a built-in state manager, requiring the use of Redux or something similar;
- Low on plug-ins: Next.js has fewer easy-to-adapt plugins available compared to other frameworks like Gatsby.js.

To conclude, below is Table 3, which considers all the information from both the React and Next.js subchapters and compiles it into a table, in order to easily visualize in a summarized way, the differences between the two. Although it's clear Next.js has generally more advantages, as it should, because that is the point of using Next.js when building React applications. There are still some areas where React wins, namely the learning curve, flexibility and popularity.

Table 3 - Comparison of features between React and Next.js

	React	Next.js
Rendering	Client-side rendering only	Supports both client-side and server-side rendering
SEO	Poor SEO due to client-side rendering	Improved SEO due to server-side rendering
Routing	Requires third-party routing solutions like React Router	Built-in routing solution
Performance	Good for small to medium-sized apps, but can become slow for larger apps	Improved performance due to server-side rendering and built-in optimizations
Scalability	Can be difficult to scale for large apps	Easier to scale due to server-side rendering
Built-in State Management	None, requires third-party solutions like Redux	None, requires third-party solutions like Redux

Learning Curve	Relatively low learning curve as it is just a library	Next.js is a full-stack framework that requires a more significant learning curve for developers who are new to it
Flexibility	React is a more flexible framework as it allows developers to choose their own stack and libraries for different components of their application	Provides a more opinionated approach to building applications
Popularity	More popular and widely used. Consequently, it has a larger pool of resources and tools available for developers	Although its popularity has been increasing in recent years, it is still less popular than React.js

4.3.3. React-Query

Tanstack developed React Query [29], a JavaScript module for handling and caching asynchronous data in React applications. It offers a straightforward and user-friendly API for obtaining and caching data from APIs, databases, and other sources. The three main features that highlight this library are fetching, caching and server state update.

Starting off with caching, using React Query, data is automatically cached in memory. In other words, future requests for the same data can be filled from the cache without requiring extra network requests. The cache can be set up to automatically refresh stale data as well, ensuring that applications are always showing the most recent data.

When it comes to data fetching and management, React Query provides a set of hooks, the most common being ‘useQuery’ (to get data) and ‘useMutation (to create, update or delete). These hooks offer a consistent and reliable way to handle data throughout applications by abstracting away the complexity of fetching and updating data.

Applications' client-side data can be in sync with the server state thanks to React Query's robust server state management capability. This feature is especially helpful in applications where client-side state must reflect regularly updated data from the server. The ‘useQuery’ hook and the ‘staleTime’ and ‘cacheTime’ parameters can be used to enable server state management in React Query. The ‘cacheTime’ parameter defines how long the data should

be stored in the cache before being erased, whereas the ‘staleTime’ option determines how long the data in the cache should be considered fresh.

To conclude this section, a quick reference to Redux, since it is also a popular alternative to React Query. Redux is a state management library [30] that provides a centralized store for managing application state. It is designed to handle complex application state that needs to be shared across multiple components and features, with a strong focus on predictability and consistency. Basically, the decision between React Query and Redux relies on the unique demands of the application. React Query is an excellent option for apps with sophisticated data fetching requirements, while Redux is a solid choice for applications with extensive state management requirements.

4.3.4. CSS – Tailwind CSS, Radix and Stitches

The presentation and design of HTML and XLM (Extensible Markup Language) documents are made possible thanks to CSS, which is a style sheet language. According to the book “CSS: The Definitive Guide” [31], the World Wide Web Consortium (W3C) first proposed it in 1996 as a means of separating presentation from content in web pages. Inline styles, font tags, and table-based layouts were the only tools available to web developers prior to the advent of CSS for modifying the visual appearance of their websites. As a result, it was challenging to make broad design changes to websites, and the HTML code was disorganized and challenging to comprehend. CSS was a significant development, which made managing and maintaining the look of a website easier by allowing the defining of styles in a separate file or in the head of an HTML document. In addition, CSS made it possible to style websites with a variety of new features that were not available with HTML alone, including background images, borders, and gradients. As CSS has evolved, more features and functionalities have been added, including animations, responsive design, and CSS grids. Due to these developments, it is now a crucial tool for web designers and developers who want to make cutting-edge, aesthetically pleasing websites.

While CSS remains the primary tool for styling web pages, there are also several alternative styling solutions available, such as Tailwind CSS, Radix and Stitches. Starting off with Tailwind CSS, it is an open-source utility-first CSS framework [32] that offers pre-defined classes for popular design trends, which make it easier to style elements without creating original CSS code thanks to Tailwind CSS. The following list showcases the three main features of Tailwind CSS:

- Pre-defined utility classes: they can be applied directly into HTML elements, for example, instead of writing custom CSS code to set the font size of a heading element, a developer can simply apply a pre-defined utility class, such as “text-xl”, which sets the font size to extra-large;
- Pre-designed components: they can be easily customised and can range from smaller components, like buttons, inputs and forms, to bigger components like pricing pages, header sections and testimonials [33];
- Responsive design: there is a variety of responsive classes that may be used to modify a website's design for various screen sizes, including mobile devices and huge desktop displays.

Another alternative is Radix, being a CSS framework that offers pre-built components and utility classes for creating contemporary, responsive websites. Similar to Tailwind CSS, it is built on a utility-first strategy, with each CSS class standing in for a single style rule. However, comparing to Tailwind CSS, Radix places greater emphasis on accessibility, making it easier to customise and extend. The key features [34] are:

- Accessibility: Radix CSS components are designed to be accessible, adhering to WAI-ARIA (Web Accessibility Initiative – Accessible Rich Internet Applications) design patterns and including support for ARIA (Accessible Rich Internet Applications) and role attributes, focus management, and keyboard navigation;
- Unstilled: Components ship with zero styles, allowing for complete control over styling. Components can be styled with any styling solution, including vanilla CSS, CSS pre-processors, or CSS-in-JS libraries;
- Opened: Radix Primitives can be altered to meet one’s requirements. There is granular access to each component part through the open component architecture, enabling to wrap any component and add custom event listeners, props, or refs;
- Uncontrolled: if applicable, components can be controlled instead of being uncontrolled by default. All the behaviour wiring is handled internally, making it easy to get up and running without needing to create local states;
- Developer experience: Radix Primitives provides a fully typed API, and all components share a similar API, creating a consistent and predictable

experience. Radix CSS also includes an “asChild” prop, giving developers full control over the rendered element.

Then there is Stitches, which is a lightweight styling library that prioritizes developer experience and component architecture [35]. With support for responsive design, theming, and CSS elements like media queries, keyframes, and global styles, it offers a potent API for building styles programmatically. Some important features to highlight are:

- Variants: Stitches treats variants as a core element for creating composable component APIs, allowing designers to define single, multiple, and compound variants for styles. These variants can be applied conditionally or responsively;
- Tokens: one of its main features is the ability to create custom tokens that can be easily applied as CSS values. Token scales are automatically mapped to CSS properties, including shorthand properties;
- Overrides: the “css” prop provided by Stitches allows for overriding styles on any Stitches components, which can come in handy for specific layout concerns;
- Utils: developers can design custom CSS Properties using their own API. This feature enables the creation of robust combinations of properties, aliases, and limitations as necessary.
- Theming: Stitches simplifies theming by providing an out-of-the-box solution. Users can generate multiple themes and apply them wherever needed. Each theme generates a CSS class name that overrides the default tokens.

Ultimately, CSS has come a long way since its introduction in 1996 and it remains a key component for styling websites, making it an indispensable web development toolkit. Some CSS frameworks have been created in order to facilitate the use of CSS, and the choice between them will depend on the project's unique requirements and the preferences of the development team. Tailwind CSS is a good choice for developers who want a utility-first approach and prefer a larger set of pre-defined classes. Radix CSS is ideal for developers who want a more modular, accessible approach with a focus on customization and theming. Stitches CSS is a great option for developers who prefer to write CSS styles in JavaScript and want a flexible, responsive solution with live preview capabilities.

4.3.5. TSX and HTML

Markup languages like TSX (Typescript XML) and HTML are both used in web development to specify the structure and content of web pages.

The most used markup language for making web pages is called HTML, which stands for Hypertext Markup Language. The foundation of each website or web application is HTML [36], a fundamental technology utilized in web development. It uses a number of tags and properties to describe elements including headings, paragraphs, links, images, and forms (for example <h1>, <p>, <link>, <img>, <form>, respectively) as well as the structure and content of web pages.

Developers can construct HTML-like code in their TypeScript files using the syntax extension known as TSX, which combines TypeScript with JSX [37]. The embeddable JSX syntax resembles XML. It is intended to be converted into legitimate JavaScript, albeit the semantics of that conversion vary depending on the implementation. TSX is used in React applications to determine the structure and behaviour of components.

Although TSX and HTML are both used to specify the structure and content of web pages, there are some significant distinctions between them, namely:

- Syntax: Unlike HTML, which has a separate section to write JavaScript (inside the <script> tag), TSX uses curly brackets to enclose TypeScript expressions that are used to dynamically generate content. HTML, on the other hand, does not support the incorporation of dynamic content in the same manner. Besides that, in HTML it’s only possible to directly type JavaScript, not TypeScript;
- React focus: TSX offers a variety of features that are exclusive to React applications, such as event handling, props of components and state management, which ultimately allows for more complex applications, whereas HTML cannot reach those features on its own;
- Type safety: TypeScript is well-known for providing type safety, which allows developers to build more reliable code and identify errors earlier in the development process. This level of type safety is not offered by HTML.

4.4.Backend

This subchapter will explore the world of server-side development using popular technologies like NestJS, TypeORM, MongoDB, PostgreSQL, and Redis. In this subchapter, we will dive into the basics of these technologies, their advantages, disadvantages, and why they are used to build scalable and performant backend systems.

4.4.1. NestJS

Nest, or NestJS, is a framework that allows developers to build efficient and scalable server-side applications using Node.js [38]. It supports TypeScript but also allows developers to write code in pure JavaScript. The framework combines elements of Object-Oriented, Functional, and Functional Reactive Programming. Nest abstracts away low-level details of HTTP server frameworks, but also exposes their APIs directly to the developer, allowing the use of third-party modules available for the underlying platform. The next paragraphs will explain core fundamentals of NestJS, such as: the NestJS CLI, Decorators, Controllers, Providers, Services, Modules, Dependency Injection and Guards.

Starting with the NestJS Command Line Interface (CLI) [39], it is a powerful tool that helps developers generate boilerplate code and manage application dependencies. It provides a set of commands that can be used to create new applications, modules, controllers, services, and providers. The CLI also helps with configuration and setup, making it easy for developers to get started with NestJS.

Decorators are commonly used throughout the applications. NestJS provides a set of built-in decorators [40] that allow developers to easily define and configure various components of their application, such as controllers, services, and providers. These decorators provide a simplified way of adding metadata to the application's classes, properties, and methods, which can be used by the NestJS framework to automatically wire up dependencies and manage the application's behaviour. Some of the most common decorators are shown in Table 4. Developers can also create their own custom decorators to extend the functionality of the framework or to encapsulate their own business logic.

Table 4 - Commonly used built-in NestJS decorators

	Usage
@Controller	Defines the base route path for all the routes declared inside a class and designates that class as a controller
@Injectable	Gives a class the ability to be injected into other classes as a dependency by designating it as a provider
@Param	Decorates and maps a parameter from the route path to a parameter in a route handler method
@Query	Decorates a route handler method parameter and converts it to a query string parameter from the request URL
@Body	A route handler method that decorates a parameter and maps it to the request body.
@Header	Decorates a parameter in a route handler method so that it can be mapped to a certain request header
@Response	Decorates a parameter in a route handler method and enables direct response object manipulation
@Request	Decorates a parameter in a route handler method and enables direct request object handling

Controllers are a fundamental building block in NestJS [41] that handle incoming HTTP requests and returns responses. Controllers are responsible for handling requests, validating inputs, and calling services to retrieve or manipulate data. In NestJS, controllers are decorated with the @Controller decorator and define routes using decorators like @Get, @Post, @Put, and @Delete. Figure 9 below, taken from NestJS's documentation referenced in this paragraph, is a high-level diagram that illustrates how requests flow from the client layers to the server layer, through HTTP, and how a certain Controller is responsible for handling that request.

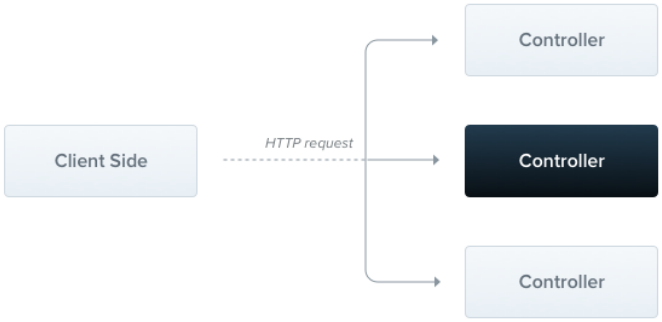

Figure 9 – High-level diagram of a Controller in NestJS

When it comes to Providers, a provider is a class in NestJS that can be injected into controllers, services, and other providers [42]. Providers are used to create and manage instances of objects that are needed by the application. In NestJS, providers are decorated with the @Injectable decorator and can be injected using the constructor of other classes. Speaking of services, a service is a class in NestJS that encapsulates business logic and interacts with data sources. Services are used by controllers to handle requests and provide responses. In NestJS, services are decorated with the @Injectable decorator and can be injected into controllers, other services, and providers.

A Module is a class in NestJS that defines a logical boundary for related components like controllers, services, and providers [43]. Modules are used to organize and encapsulate functionality in an application. In NestJS, modules are decorated with the @Module decorator and can be imported and exported by other modules. Figure 10, taken from NestJS's documentation referenced in this paragraph, shows how modules are arranged in a hierarchical structure, with the root module at the top and child modules nested within it. Each module can contain controllers, providers, and other modules, and can import and export functionality to and from other modules in the application. The diagram also highlights the concept of dependency injection, which is a design pattern that is used extensively in NestJS to manage dependencies between components like controllers, services, and providers. In NestJS, dependencies are injected into classes using the constructor, making it easy to swap out implementations or mock dependencies for testing

purposes. Taking the example of the figure below, the Orders Module and Chat Module are importing the functionalities of Feature Module 1, 2 and 3, according to the Figure 10.

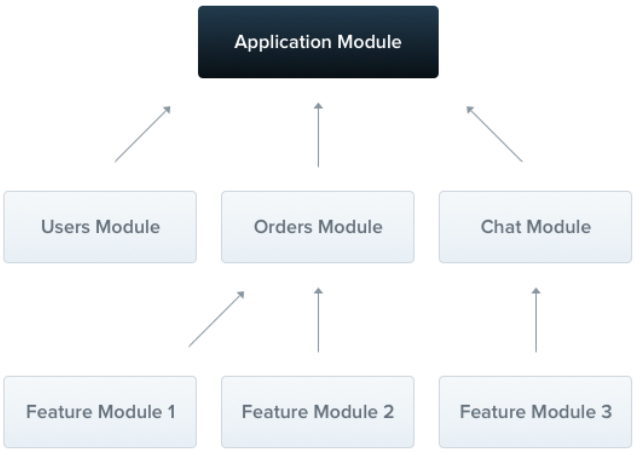

Figure 10 - Example organization of Modules in NestJS

Last, but not least, Guards. Guards are a powerful feature in NestJS that are used to protect routes by validating incoming requests [44], as seen in Figure 11, once again taken from NestJS's documentation. Guards can be used to implement authentication, authorization, and input validation. In NestJS, guards are implemented as middleware and can be attached to routes using decorators like @UseGuards.

Figure 11 - Diagram showcasing the role of a Guard, in NestJS

Nest.js includes a comprehensive built-in testing module that provides a wide range of testing utilities [45]. The testing module includes features such as mocking dependencies,

testing database connections, and parallel test execution, which makes it easier to write and run unit tests for Nest.js applications. The built-in testing module also provides support for end-to-end testing tools. Additionally, Nest.js has integration with popular testing libraries such as Jest. By leveraging the built-in testing module, developers can write reliable, maintainable, and scalable tests to ensure the quality of Nest.js applications.

In order to determine the advantages and disadvantages of using NestJS as a backend framework, there is a Bachelor thesis focused specifically on that topic [46]. The thesis gathered information by not only reviewing data from other researchers and books, but also by interviewing four experienced developers that worked with NestJS, proving the reliability of the study.

The conclusion the study came up to, regarding the advantages, are the following:

- Highly opinionated framework, having Angular-like architecture;
- The CLI is a great tool for developers to quickly generate projects, files, tests, boilerplate code, and settings;
- High scalability;
- Automated testing starter makes testing easier.

As per the same study, the major drawbacks of using NestJS are:

- Lack of documentation;
- Limited community support;
- Steep learning curve for new developers.

4.4.2. TypeORM

According to NestJS's official documentation [47], TypeORM is considered to be "the most mature Object Relational Mapper (ORM) available in the Node.js world". It is a library for TypeScript and JavaScript and is designed to simplify database operations by mapping database tables to classes and vice versa.

TypeORM's official website [48] specifies that numerous databases, including PostgreSQL, MySQL, MariaDB, SQLite, and others, are supported by TypeORM. It offers a vast array of functions, including the ability to create database tables, insert, update, and delete records, perform migrations, and establish relationships between tables. Developers can select the most suitable approach for their project because TypeORM supports both the

Active Record and Data Mapper designs. Additionally, it works well with other frameworks like Node.js, React, and React Native. The next two paragraphs will discuss the two concepts that every developer must know when developing an application that involves TypeORM: Entities and Relations.

Starting with Entities [49], a database table is represented by a class called an entity. Each instance of the class corresponds to a record in the database, and its properties correspond to table columns. By using decorators to define properties, indexes, relationships, and restrictions, TypeORM enables developers to create entities. Additionally, it comes with built-in support for entity listeners, embedded entities, and inheritance. Entities allow for quickly and easily data manipulation using object-oriented programming techniques, while the ORM handles transforming the objects to SQL statements. The TypeScript code in Figure 12, taken from the same TypeORM Entities documentation, defines a User entity in TypeORM, which represents a database table with columns for “id”, “firstName”, “lastName”, and “isActive”. The @Entity() decorator is used to mark the User class as an entity, and the @PrimaryGeneratedColumn() decorator is used to specify that the id column is a primary key with auto-increment. The @Column() decorator is used to specify the other columns. When the application runs, TypeORM will generate the corresponding database table with the specified columns and properties.

```ts
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    isActive: boolean
}
```

This will create following database table:

```
+-------------+--------------+----------------------------+
|                         user                            |
+-------------+--------------+----------------------------+
| id          | int(11)      | PRIMARY KEY AUTO_INCREMENT |
| firstName   | varchar(255) |                            |
| lastName    | varchar(255) |                            |
| isActive    | boolean      |                            |
+-------------+--------------+----------------------------+
```

Figure 12 - Example of an Entity, from TypeORM documentation

Another important aspect of TypeORM is its support for relations between entities [50]. Using decorators like @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany, TypeORM enables developers to specify relationships between entities. It is possible to set cascading actions like insert, update, and delete as well as how eagerly or slowly the relations are loaded. Additionally, TypeORM allows composite keys, foreign keys, and join tables. Developers can easily query related data with TypeORM, and the ORM will create the proper SQL statements to rapidly extract the data. An example of a relation between entities can be seen in Figure 13, which was taken from the Relations documentation referenced in this paragraph. The given code defines two entities in TypeORM, Category and Question, and establishes a Many-to-Many relationship between them. The Category entity has an “id” column, a “name” column, and a “questions” property that represents an array of Question entities. The Question entity has an “id” column, a “title” column, a “text” column, and a “categories” property that represents an array of Category entities. The @ManyToMany() decorator is used to specify the many-to-many relationship between the entities, and the @JoinTable() decorator is used to create a join table in the database to manage the relationship. The cascade option is set to true to enable cascading of changes to related

entities (in other words, if we create a Question record with categories that do not exist in the database, those categories will automatically be created). This code demonstrates how TypeORM allows developers to define and manage relationships between entities in a convenient and intuitive way.

```typescript
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany } from "typeorm"
import { Question } from "./Question"

@Entity()
export class Category {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    name: string

    @ManyToMany((type) => Question, (question) => question.categories)
    questions: Question[]
}
```

```typescript
import {
    Entity,
    PrimaryGeneratedColumn,
    Column,
    ManyToMany,
    JoinTable,
} from "typeorm"
import { Category } from "./Category"

@Entity()
export class Question {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    title: string

    @Column()
    text: string

    @ManyToMany((type) => Category, (category) => category.questions, {
        cascade: true,
    })
    @JoinTable()
    categories: Category[]
}
```

Figure 13 - Example of a Relation between Entities, from TypeORM documentation

Overall, TypeORM is a strong and adaptable ORM framework that makes working with databases easier and simpler.

4.4.3. MongoDB

MongoDB started being developed in 2007 by a start-up called “10gen” [51], and since its first release in 2009, MongoDB has become a well-known open-source NoSQL document-oriented database system. Modern online applications frequently use it because of its scalability, performance, and reputation for flexibility.

As previously stated, MongoDB, as a document-oriented NoSQL database system, stores data as documents as opposed to tables. It allows for developers to store data in a flexible manner that meets the needs of their applications, unlike with conventional relational databases that demand a specific structure. BSON (Binary JSON), which is a binary-encoded serialization of JSON-like data, is the format used by MongoDB to store documents. Arrays, embedded documents, binary data, and other data types can all be used in many fields inside a single document. With the help of this flexible data model, programmers may describe complex, hierarchical data structures in a way that feels intuitive and natural.

MongoDB's versatility is one of its main benefits. Developers can store data in a way that makes sense for their application because of the flexible schema it uses. Hence, a wide variety of data formats and structures, such as JSON, arrays, and embedded documents, can be handled by MongoDB. In addition, MongoDB is highly scalable and adept at managing huge traffic and data quantities. Performance is another benefit of MongoDB. It makes use of a memory-mapped storage engine, which enables quick read and write rates and is excellent for applications that need high throughput. Furthermore, MongoDB offers a variety of replication options, including master-slave and sharded replication, making it the perfect choice for programs that need high availability and data redundancy.

When it comes to interacting with MongoDB databases, Mongoose is a popular choice [52]. It is a ODM (Object Data Modelling) library for Node.js that makes it easier to interact with MongoDB. The Node.js driver provides low-level access to MongoDB, and while it offers more flexibility, it also requires more code and can be more error-prone than using Mongoose, in tasks like data validation and schema creation. Mongoose, on the other hand, provides a simpler and more intuitive API that abstracts away many of the complexities of working with MongoDB, with the counter point that in some cases it can be less flexible. The main features of Mongoose are the following [53]:

- Data modelling and validation: the MongoDB documents can have its structure defined using schemas, which provide a variety of data types, like strings,

numbers, Booleans, dates and arrays. The structure can be validated using the built-in validation methods, provided by Mongoose;

- Query Building: a chainable query builder API allows for complex queries to be built, while maintaining a natural syntax. This API makes it easier to sort, filter and paginate data;
- Populate: it gives the possibility to populate a document with data from other collections in the database. By doing so, nested searches can be avoided or the amount of data that needs to be loaded into memory may be decreased.
- Middleware: MongoDB queries can be modified before and after the execution of asynchronous functions at the schema level.
- Virtuals: it corresponds to the creation of "virtual" properties on documents, which are calculated from other properties or database data. Code can be made simpler by using these virtuals, and less data needs to be kept in the database.

While MongoDB has many advantages, it is also important to identify its weaknesses [54]. One of the disadvantages of MongoDB comes down to reliability. By default, MongoDB writes are asynchronous. The key benefit of this is that there is no need to wait for confirmations before beginning any insert or update operation. Updates are quicker as a result, but less trustworthy. The write operation will still partially succeed even if some of the updates are failed. Another important disadvantage to keep in mind is the fact that MongoDB's document-oriented approach makes maintaining data integrity and consistency in a variable schema environment more difficult. Also, handling complex joins and relationships between documents, which are simpler in a relational database, can be more challenging.

4.4.4. PostgreSQL

From its introduction in 1996, PostgreSQL, also known as Postgres, has been a potent open-source relational database management system (RDBMS), considered by many to be the most advanced open-source database system [55]. It is a popular option for many applications because of its cutting-edge capabilities, scalability, and high level of extensibility. Unlike other RDBMS systems, PostgreSQL has a distinctive architecture. By utilizing a multi-process design, it offers a high level of concurrency and can manage several connections simultaneously. Moreover, it has a modular architecture, which makes it

extremely scalable and configurable and enables developers to add or remove features as needed.

The fact that PostgreSQL supports ACID (Atomicity, Consistency, Isolation, and Durability) transactions, which guarantee data consistency, durability, and isolation, is one of its main benefits [56]. It is therefore a trustworthy and safe option for applications that need transactional support. PostgreSQL is a flexible option for a variety of use cases because it additionally supports a wide range of data types, including JSON and GIS (Geographic Information System) data. Besides that, another point in its favour is its wide range of advances features, making it a popular choice among many types of applications. For example, it is excellent for applications that call for sophisticated search capabilities because it comes with built-in full-text search functionality. Furthermore, PostgreSQL offers both synchronous and asynchronous replication, making it the perfect choice for applications that need high availability.

In terms of disadvantages, it comes down to two main things: complexity and extreme performance. While on one hand, having advanced features is a great perk, on the other hand, it can make it difficult to set up, configure, and learn all possible features. In spite of its high extensibility, there may be compatibility problems when utilizing third-party extensions. When it comes to performance, despite being known for its scalability and reliability, PostgreSQL might not always be the fastest option. High-write applications could be better suited for other databases like MongoDB or Cassandra.

4.4.5. Redis

Redis is an open-source [57], in-memory data structure store that is utilized as a message broker, database, and cache. Because it supports a variety of data structures, including strings, hashes, lists, sets, and sorted sets, it is frequently referred to as a "data structure server". Redis is built with scalability and excellent performance in mind. It can read and write data incredibly quickly because all of the data is stored in memory. Redis also offers persistence, which means that data can be saved into disk periodically or on-demand to prevent data loss in case of system failure.

Caching is one of Redis' primary use cases [58]. Redis helps speed up web applications by reducing the amount of costly database queries by keeping frequently visited data in

memory. Moreover, Redis provides sophisticated caching capabilities including invalidation, eviction, and expiration.

Besides caching, Redis is frequently used as a queuing system due to its rapid performance and integrated queue support. Lists and sorted sets are only two examples of the queue-supporting data structures that Redis offers. Redis commonly implements queues as lists of items, where new items are added to the end of the list and old items are removed from the head of the list. This is known as a "first in, first out" (FIFO) queue. Other queueing patterns, like priority queues, in which items are dequeued in accordance with their priority level, are also supported by Redis. Redis' capacity to manage massive amounts of data and requests is one of the advantages of utilizing it as a queuing system. Due to its ability to process millions of requests per second, Redis is perfect for high-traffic web applications that need quick and dependable queuing.

Redis can also be used as a message broker for real-time applications [59]. Subscribers are able to join and leave at any time. The Publish/Subscribe technique, also known as Pub/Sub, makes data available while it is being subscribed. There is no mechanism in Pub/Sub to acquire "older" values or to re-request values. Redis' ability as a data store, however, allows it to offer these functions.

Although its benefits, Redis still has some disadvantages that cannot be ignored, for instance [60]:

- Data storage: Redis provides a variety of solutions for data persistence (disk being the main one), but data loss is still a possibility in some cases, for example a process crash or restart. Because of this, Redis is a poor choice for applications that demand high standards of data dependability and longevity;
- Limited Data Structure Support: Redis only supports a limited set of data structures such as strings, hashes, lists, sets, and sorted sets. This means that it may not be the best option for use cases that require more complex data structures;
- Limited Querying Capability: Redis does not support complex querying capabilities, which means that it may not be the best option for applications that require advanced querying features.

4.5.Methodologies

It is essential for teams to embrace flexible and effective methodologies in today's fast-paced and dynamic software development environment in order to produce high-quality products. This subchapter will explain some of the most well-known Agile approaches, like Scrum and Kanban, as well as the best practices of Continuous Integration and Continuous Delivery, also known as CI/CD, which help teams accelerate customer value delivery by streamlining their development processes.

4.5.1. Agile and CI/CD

Agile Methodology is an approach to project management and software development that places a high value on flexibility, teamwork, and quick iterations [61]. According to the 2001 Agile Manifesto [62], working software and customer collaboration take precedence over in-depth documentation and restrictive processes. A key aspect of Agile are its 12 Agile principles, which are a set of guidelines that serve as the foundation for the Agile development approach. These guidelines place a strong emphasis on communication, cooperation, adaptability, and quick iteration. The guiding principles encourage Agile teams to put working software first, welcome change, and keep a manageable work tempo. Also, they emphasize the value of direct communication, self-organizing teams, and routine reflection and adaptation. Generally, the Agile principles encourage a client-centred, team-based, and flexible method of software development. Besides the principles, the Agile Manifesto also outlines 4 Agile values, which are:

- Individuals and interactions over processes and tools;
- Working software over comprehensive documentation;
- Customer collaboration over contract negotiation;
- Responding to change over following a plan.

Another critical aspect of the Agile methodology is CI/CD [63]. It refers to the automation of building, testing and deploying code changes. Agile teams may quickly and safely deliver new features and updates to users through an automated pipeline with the help of CI/CD. By identifying problems early in the development process and enabling quick iteration and feedback, this method helps to decrease errors and boost efficiency. Teams can respond swiftly to changing market conditions and client needs thanks to CI/CD.

4.5.2. Scrum

Among many agile methods, Scrum is the most popular approach [64] for creating software products. Scrum is essentially a flexible, agile framework that outlines actions to manage and direct the creation of software and other products. One of Scrum's key aspect is that is consists of three roles:

1. Product Owner, who defines the product backlog, which is a list of features, improvements, and bug fixes that need to be made;
2. Scrum Master, whose main duties include aiding the Scrum process, removing obstacles, and assisting the team in getting better;
3. Development Team, which at the conclusion of each sprint, is responsible for providing a potentially shippable product increment.

Another key point of Scrum are sprints. An iteration that is time-boxed and usually lasts between one and four weeks is called a sprint. Throughout the sprint, the Development Team works on a number of features from the product backlog, and a sprint backlog is used to monitor progress. The Development Team holds daily Scrum meetings during the Sprint to review their advancement, daily plans, and any challenges they may be encountering. The Development Team presents the finished features to the Product Owner in a sprint review meeting at the end of each sprint. Based on the input and priorities, the Product Owner then chooses which features to include in the following Sprint. Scrum places a strong emphasis on empirical process control, which calls for regular reviews and adjustments of the team's progress. Via transparency, inspection, and adaptation, the Scrum methodology enables teams to continually improve and produce high-quality products.

4.5.3. Kanban

Kanban is an Agile methodology that focuses on 4 main points: visualization of work, limitation of work in progress, focus on flow, and continuous improvement. It was originally developed by Shigeo Shingo, a Japanese JIT (Just in Time) authority and engineer at Toyota Motor Company [65].

The Kanban methodology makes use of a visual board to depict the workflow, with each column standing for a different stage. Typically, the board is divided into "To-Do," "In Progress," and "Done" columns. Cards or sticky notes that represent the work items are moved through the columns as they are processed. The team can comprehend the current

state of the project and identify process bottlenecks with the aid of the visual board, which also promotes transparency. While simple operations could be accomplished using this method, most business processes in the actual world involve a number of phases. It would be difficult to manage and optimize your workflow with just one “In Progress” column because it is difficult to see the processes that are actually taking place during this condition. Kanban cards must at the very least include the task name and the names of the team members assigned to work on the card. In some cases, it may also include estimations of the amount of effort, such as the number of hours. A unique identification is typically used so that it is simple to distinguish between cards and avoid having to look up long job titles. Figure 14 is an example of a Kanban board used in a real-world scenario, more specifically for the CMD Center project, which I worked on during the internship. There are 4 columns, “To Do”, “In Progress”, “In Review” and “Closed”, which are self-explanatory, and each card represents an issue/task, which has the following structure: an identifier between square brackets, followed by the title; a set of labels to categorize the card; the ID of the issue/task of that card; an avatar of the person assigned to that card.

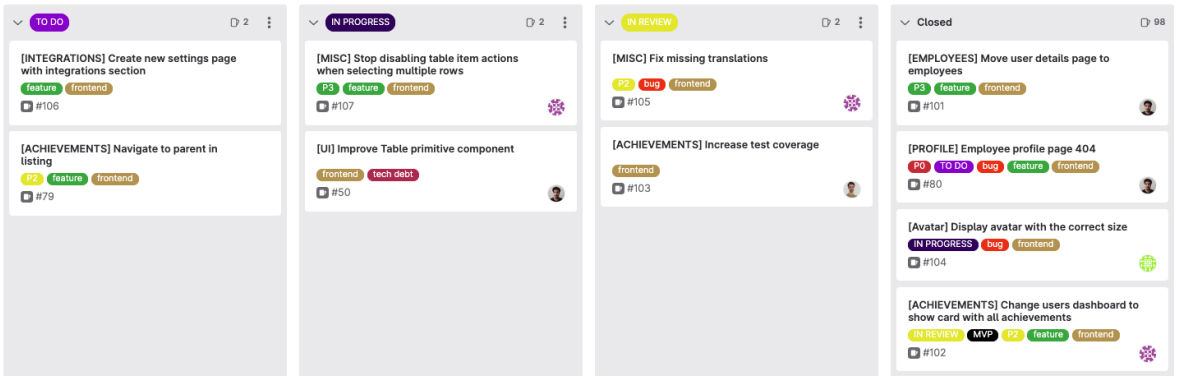

Figure 14 - Snapshot of Gitlab's Kanban board from the CMD Center project

Kanban places more emphasis on the "pull" rather than the "push" paradigm [66]. In other words, rather than being assigned in advance, work is only added to the system when the team is prepared to begin working on it. By doing so, the workflow is optimized and the amount of work in progress (WIP) is decreased. Kanban emphasizes continuous improvement as well. The team evaluates the procedure on a regular basis and pinpoints potential improvements. They evaluate the efficiency of the process and spot bottlenecks using indicators like lead time, cycle time, and throughput. The team may streamline workflow and provide value to customers more quickly by iteratively improving the process.

4.5.4. Scrumban

According to the Agile Alliance [67], Scrumban is a hybrid agile methodology that combines the principles of Scrum and Kanban. Scrum is an iterative and incremental framework for managing complex projects, while Kanban is a lean methodology focused on continuous delivery and minimizing waste. Scrumban aims to combine the strengths of both frameworks to create a more flexible and efficient approach to software development. In Scrumban, work is managed through a backlog of prioritized user stories or tasks, and teams work in short, time-boxed iterations or sprints, similar to Scrum. However, unlike Scrum, Scrumban allows for changes to be made to the backlog and priorities during the iteration based on feedback and evolving business needs. In addition, Scrumban incorporates the visual management and flow principles of Kanban, with work items visualized on a board and moved through different stages of completion. This allows for greater transparency and collaboration within the team and helps identify bottlenecks and areas for improvement.

Overall, Scrumban can be a useful methodology for teams looking for a more flexible and adaptable approach to software development, that still incorporates the structure and discipline of agile methodologies like Scrum.

4.5.5. Pair Programming

Pair programming [68] is a collaborative programming technique where two programmers work together on the same task using a single computer. One person, the "driver," is responsible for writing code, while the other, the "navigator," is responsible for reviewing the code in real-time, searching for errors, and suggesting solutions. Pair programming has been around for decades but has become more popular with the rise of agile methodologies and XP (Extreme Programming). It can be used throughout the entire software development process as it has shown to improved product quality.

Over the years, studies have shown [69] that this methodology has many benefits, but it still is important to keep in mind its problems. Starting with the benefits, the most commonly cited benefits are:

- Fewer bugs (higher code quality): pair programming helps identify and fix bugs early in the development process, which leads to better code quality;
- Increased code understanding: it facilitates knowledge sharing and prevents the risk of having only one person with expertise in a particular codebase;

- Ability to learn from a partner: developers can learn from each other and promotes better design and implementation through collaboration and brainstorming.

When it comes to the problems, according to the same reference as the benefits, these are the most common:

- Cost: the main concern for organizations is the cost, as two people are being paid to do the work of one. There is also scepticism that having two people work on one task is a good use of resources;
- Scheduling: time to work in pairs is the second most cited problem, as the two partners require equivalent schedules, which makes it difficult to block out two calendars;
- Personality clash/disagreements: finding compatible partners can be difficult, as it is hard to find pair programmers that have compatible personalities, value systems, and lifestyles.

In summary, pair programming can be a valuable technique for improving code quality, team communication, and knowledge sharing. However, it may not be suitable for all teams or situations, mainly because of the cost involved, since technically there are two people working on the same thing.

4.6.Summary

In the State of the Art section, a comprehensive overview of the technologies, libraries, frameworks, and methodologies employed in the internship is presented. It starts by giving an overview about the architecture of the Web. The second subchapter focuses on the programming language used throughout the entire internship, Typescript. It is a versatile and robust programming language that combines the flexibility of JavaScript with the added benefits of static typing, enabling enhanced code reliability and maintainability. It was used on the frontend of all three projects I worked on.

The third subchapter is related to the frontend technologies. React, a popular JavaScript library, provides a component-based approach to UI development, allowing for reusable and modular code. Next.js, a framework built on top of React, adds new capabilities to React, such as: server-side rendering, seamless file-based routing, API routes, and optimized

bundling. React-query enhances data fetching and caching, while CSS frameworks like Tailwind, Radix, and Stitches aid in creating visually appealing and consistent designs.

The fourth subchapter explores the backend technologies. NestJS, a progressive Node.js framework, provides a solid foundation for building scalable and efficient server-side applications. TypeORM simplifies database management by offering an object-relational mapping approach, supporting both MongoDB and PostgreSQL databases. Furthermore, the integration of Redis, a robust in-memory data store, empowers the backend system with advanced queuing capabilities.

Finally, the fifth subchapter dives into the methodologies adopted throughout the internship. The Agile methodology, known for its iterative and incremental approach to software development, serves as the overarching framework. Within the Agile framework, various methodologies such as Scrum, Kanban, and Scrumban are key to effectively manage a project lifecycle, promote collaboration, and ensure efficient task allocation and tracking. Additionally, pair programming is used to enhance code quality and foster knowledge sharing among team members.

5. Project 1 – SPLIT

After one month and a half of training and onboarding into the company, I joined a team of developers responsible for the development of a project called “SPLIT”. It is a retrospective tool started by xgeeks in September of 2021, that began being used by the company in January of 2023. I was part of the project from 14th of November until 6th of January. The next paragraphs will give an overview to what are retrospectives and why a tool like SPLIT made sense to be created.

Retrospectives are scheduled sessions during which the teams evaluate the previous Sprint, draw lessons from the process, and make improvements for the upcoming Sprint. During the retrospective process, the main topics that need to be addressed are:

- What went well;
- What needs improvement;
- Actions points.

One of the Agile Manifesto principles is addressed by retrospectives, which is: “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly” [62]. However, doing retrospectives when the number of people involved is large (in the case of xgeeks over 100 members) is not scalable because it requires a lot of effort put into organisation and management. Combined with the fact that there is no free and open-source retrospective tool on the market, the idea to create a tool like SPLIT came up and was followed through.

SPLIT is a free and open-source retrospective tool [70] designed for large teams or company-wide retrospectives. It does so by automatically dividing the participating group into smaller teams and electing one responsible per team that will be that sub-team representative on the main retro event. Some other key aspects are the SSO (Single Sign-On) login through Microsoft, something that the company uses in a lot of its tools, and Slack integration, in the sense that in every retro, SPLIT will create the necessary slack channels for each team.

Given that the company gives a big emphasis to feedback as a way to improve and evolve, in the last week of each month a retrospective is scheduled to take place. It consists of four main phases:

1. Random teams are created, and each member can create as many cards as they want to;
2. Team responsibles schedule a meeting in case there are cards to discuss and merge them into the main board;
3. Voting phase, where everyone in the company is notified by the Slack bot to vote in the cards;
4. Stakeholders and responsibles meet for approximately 30 to 60 minutes to discuss the cards and create the actions points.

5.1.Related Work

Before the creation of SPLIT, xgeeks used EasyRetro [71] as a retrospective tool. However, because it was not suited for large teams (100 plus in the case of xgeeks), there were some challenges that came with the use of EasyRetro, such as:

- All the members of the company needed to be manually added to a private slack channel for the retrospective;
- The channel's participants were divided up into smaller teams using a random team generator, and each team was given a responsible;
- If a person could not be responsible, a new responsible needed to be chosen, manually;
- Every responsible needed to create their teams board, share the link in the slack channel and merge all cards into the main board, all of this manually;
- Maintaining a file used in the team’s generator in order to exclude stakeholders and new joiners (employees with less than three months in the company) to be chosen as responsible.

Besides EasyRetro, there are other alternatives for retrospective tools, such as TeamRetro [72] and Retro.io [73], for example. Below is a comparison table between SPLIT and other retrospective tools. Table 5 compares the free version’s features of each tool. The reason for not comparing the paid features is because SPLIT is a free tool, so it makes sense to compare it to other free tools.

Table 5 - Retrospective tools comparison (free version)

	SPLIT	EasyRetro	TeamRetro	Reetro.io
Open Source	✓	✗	✗	✗
Pricing	Free	Free, Paid	Free Trial, Paid	Free, Paid
Integrations	Slack	✗	Slack, Trello, Asana	✗
Board Merging	✓	✗	✗	✗
Number of Boards	Unlimited	Unlimited	3	Unlimited
Number of Teams	Unlimited	Unlimited	3	Unlimited
Sub-boards	✓	✗	✗	✗
Sub-teams	✓	✗	✗	✗
Voting System	✓	✓	✓	✓
Anonymous Feedback	✓	✓	✓	✓

In summary, SPLIT is the best choice for a free tool not only because it provides the most features, but also because it allows big teams to be split into various sub-teams, each with a sub-board, and then merge all the sub-boards into the main board, which is a lot easier to manage for a big company/team. Besides that, it is the only open-source tool, making it an important factor for teams looking for a tool that can be customizable and tailored according to their needs, which is the main reason xgeeks decided to develop it. On the other hand, if cost is not a problem, then the other retrospective tools offer more features in their paid plans, such as analytics, integration with more tools and priority support.

5.2.Methodology

The development for this project followed an Agile methodology with a focus on Scrum, that enabled us to adapt to changing requirements, deliver value incrementally, and foster effective teamwork throughout the project lifecycle.

To manage our user stories and tasks, we utilized GitHub as a central platform. User stories were written and fragmented into smaller tickets, which provided a clear breakdown

of the work required. This allowed us to prioritize and track progress effectively, ensuring that all team members were aligned on the project's objectives and tasks at hand.

One of the key aspects of our Agile implementation was the use of daily meetings, also known as stand-ups. These meetings occurred five times a week, at 9:30 in the morning, providing an opportunity for the team to synchronize, discuss progress, and identify any obstacles or challenges. Additionally, we also held a bi-weekly meeting, where previous team members could join in and give their feedback on the current state of the application, as well as plan the next important steps/feature to make.

Furthermore, throughout the six-week duration of the project, I actively engaged in pair programming with a fellow colleague who were also interns. Pair programming facilitated knowledge sharing, accelerated problem-solving, and fostered a strong collaborative dynamic between us. Working closely together, we leveraged each other's strengths, learned from one another, and contributed to the overall success of the project.

5.3.Architecture

Starting with the frontend of the application, as seen in Figure 15, it was built with Next.js, a React framework. Just like every project I was part of it used TypeScript as the programming language. However, when it comes to styling, this project used Stitches and Radix as the CSS frameworks, as opposed to the other two projects which used Tailwind.

When it comes to the backend, it was built with NestJS, a Node.js framework for building efficient and scalable server-side applications. The backend had integrations with other services, namely Redis, the Slack API, the MongoDB database, and the Microsoft authentication.

The open-source version of the application allowed for both email/password authentication, but the version that was used internally by the company only allowed for Microsoft authentication. The reason was due to the fact that every employee had a company email from Microsoft, making the authentication more consistent. The authentication of the application used OAuth 2.0 [74], which is a secure authorization framework that allowed the project to authenticate and access services on behalf of the users without sharing their login credentials. As a note, OAuth 2.0 was used on all projects I was part of during the internship.

Redis was employed as a queueing system to manage jobs such as sending Slack messages and handling database queries. This allowed for efficient task scheduling, execution, and coordination, enhancing the overall performance and responsiveness of the application.

The database for this project was MongoDB, a non-relational and document oriented database. Its flexible document-oriented approach facilitated seamless handling of complex data models.

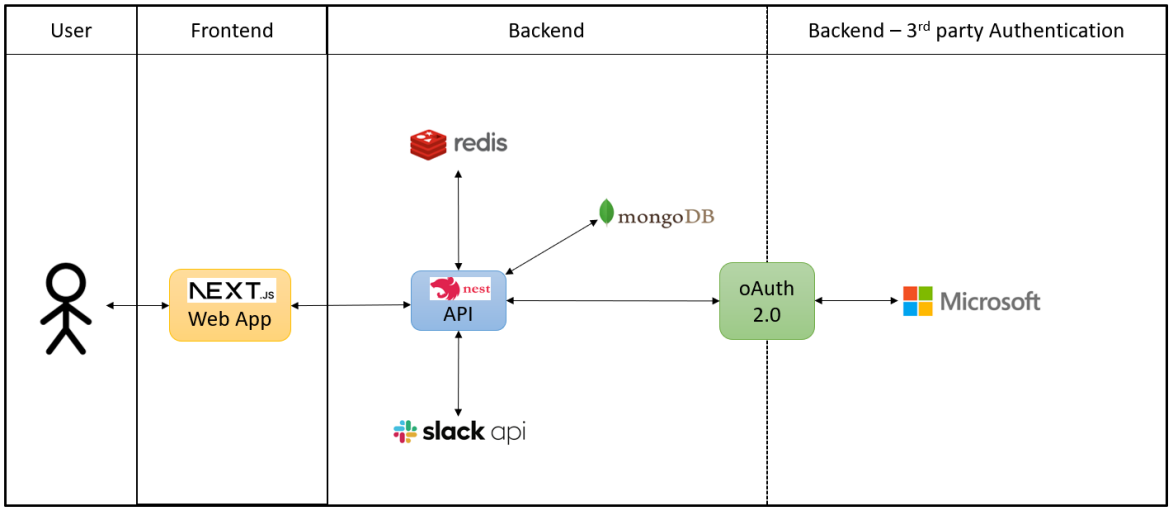

Figure 15 - SPLIT Applicational Architecture

5.4.Development

For the entire duration I was part of the project, I was assigned the role of a Fullstack developer, meaning that I worked on both the frontend and backend of the application. The infrastructure of the project was not of my concern to work on, which can be said for all projects I was part of. All parts of the application were almost ready for the first product release, with the exception of user management, which the development hadn't even started. With that said, I was assigned the task to start and finish the user management feature.

The user management feature translated into two pages: users list and user edit. The layout and features of these pages changed according to the user role (Super Admin or not). The users list page must've had limiting features, and the edit page should only be allowed to Super Admins.

The features for this project were created in GitHub issues section. Below in Figure 16 can be seen an example of some issues and can be noticed a pattern that must have always been obeyed, which is the format of the issue needs to start with the type of issue inside square brackets. Some of the most common types of issues were the following:

- [FEAT]: adding a new feature;
- [BUG]: for fixing bugs;
- [REFACTOR]: neither adds, changes nor removes a feature, it's simply a code refactor.

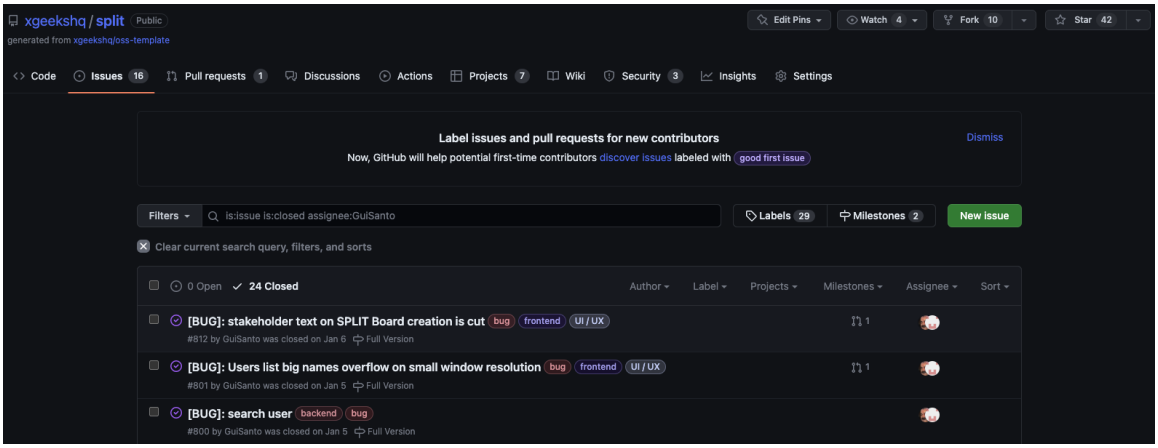

Figure 16 - SPLIT GitHub issues example (SPLIT)

As mentioned in the previous paragraph, there was a structure in the issues created, and same goes for the commits and every PR (Pull Request). Every time a PR was opened, reopened, or edited, a pipeline which was setup using GitHub Actions would always run. That pipeline, in Figure 17, consisted of two workflows:

- Test & Build: consisted of five jobs, responsible for installing the dependencies of the project, building, and testing the frontend and backend;
- Validate PR: only one job, to validate the title of the PR. The validation checked if the title of the PR was equal to the title of the last commit, as well as the format. The format always had to be the type of the issue, followed by colon with a less than 50-character description.

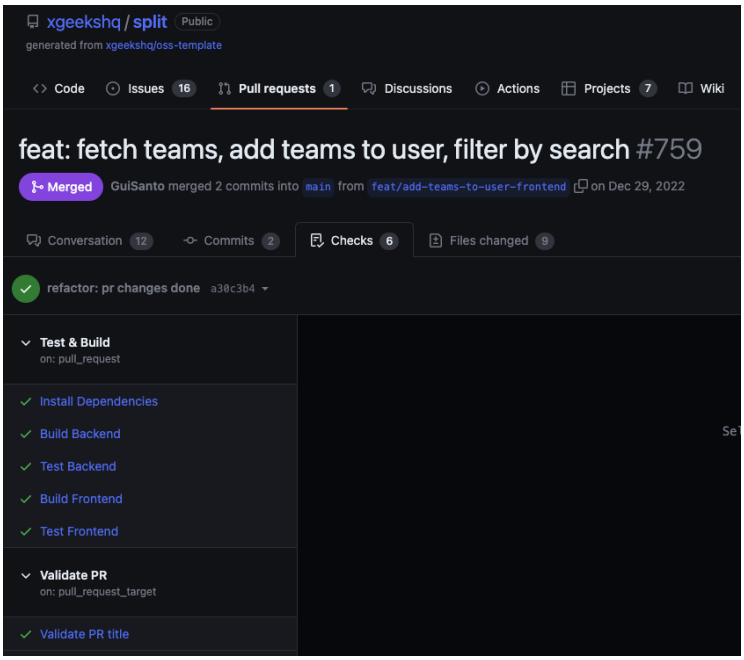

Figure 17 - GitHub Actions workflows for a sample PR (SPLIT)

An example of a wrong commit message can be seen below, in Figure 18. At first glance, it appears to be in the right format, and it is. However, the only thing that is wrong is the type, which in this case should be “feat” and not “feature”.

Figure 18 - Example of an invalid commit message (SPLIT)

The next two subchapters will explain the two user management features I developed: the user list and the edit user.

5.4.1. User List

Starting with the users list page, Table 6 shows all the features of that page. Those features did vary according to the user role, in this case if the user was a Super Admin or not. Basically, if a user was not a Super Admin, all he could do was read the information, as opposed to a Super Admin that had privileges to change it.

Table 6 - Users list page features according to user roles (SPLIT)

	Super Admin	Other
Total number of registered users	✓	✓
Users filter by name and email	✓	✓
List of all users with lazy loading	✓	✓
Each user card must have information about name, email, number of teams, is Super Admin	✓	✓
Show Super Admin role	✓	✓
Change Super Admin role	✓	✗
Edit button	✓	✗
Delete Button	✓	✗

SPLIT, as well as all projects I was part of, had their design made in Figma, a collaborative design tool. The outcome of the users list page can be seen in Figure 19.

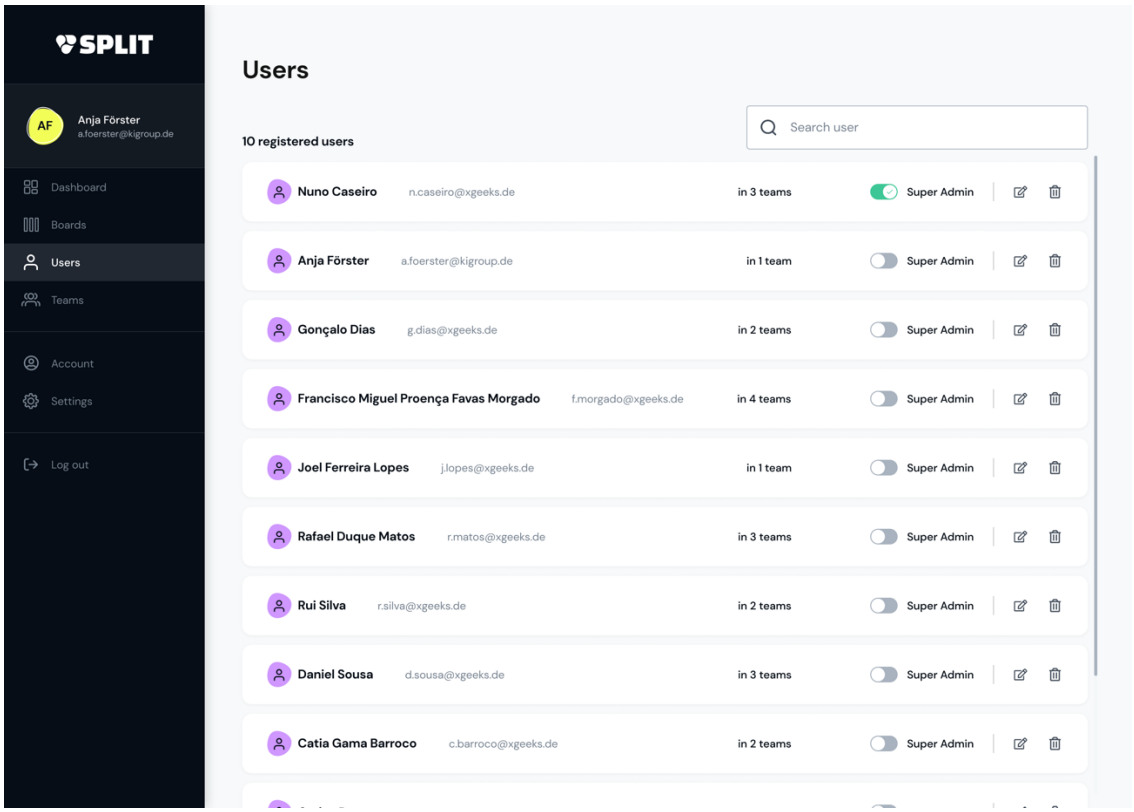

Figure 19 - Users list page (SPLIT)

When it comes to the code of this page, it can be seen below in Listing 1. In the left part of the figure, it shows the folder structure of a typical Next.js application, which will translate to the URL pathnames of the application. In this case, for the users list page, it corresponded to "/users" pathname. The code utilized the "useSession" hook to retrieve the user's session and only renders the page if a session is present (in other words, if the user is authenticated). The page structure was wrapped in a Flex container and includes a Suspense [75] fallback component for loading, which displayed a loading component while the children was loading its content. It also had a QueryError component for handling any errors related to "useQuery" data fetching. The UsersList was the component that contained the various components that made up the users list page.

```
import { Suspense } from 'react';
import { useSession } from 'next-auth/react';
import QueryError from '@/components/Errors/QueryError';
import LoadingPage from '@/components/loadings/LoadingPage';
import Flex from '@/components/Primitives/Flex';
import UsersList from '@/components/Users/UsersList';

const Users = () => {
  const { data: session } = useSession({ required: true });

  if (!session) return null;

  return (
    <Flex direction="column">
      <Suspense fallback={<LoadingPage />}>
        <QueryError>
          <UsersList />
        </QueryError>
      </Suspense>
    </Flex>
  );
};
export default Users;
```

Listing 1 - Users list page (SPLIT)

The UsersList component was responsible for displaying the registered users in SPLIT. It made use of React Query for data fetching. The component fetched users with teams from an API using an infinite query [76]. Infinite Queries allow for seamless pagination and dynamic loading of data by automatically fetching the next set of results as the user scrolls or performs actions, enabling lazy loading. Lazy loading, in my opinion, was a great feature to have because:

- the users list was quite big at the time (100 plus) and the tendency is to grow larger over time;
- the user could only see 10 to 15 users in the list at the same time, on a typical 16-inch screen, which corresponds to most use cases. Because of this, the query was set to retrieve 15 users per request, which is triggered every time the user scrolls to the end of the list;
- the page load became faster, because the app would only have to load 15 users at a time, instead of 100 plus in one go, improving the user experience.

It also included a search input to filter the users based on the entered search term. The list was rendered in a scrollable container, and additional users were loaded when scrolling to the bottom of the list. Error handling was implemented to display a toast message in case of an error during the data fetching process. The component also displayed a loading spinner while the data was being fetched.

The data was being fetched from the endpoint "/users/teams". That corresponded to the User's module, in our NestJS backend. Each module had a controller, and in this case, the User controller had an endpoint that handle any request made to the "users/teams" API endpoint. The endpoint could receive three optional parameters:

- page: number of the page;
- size: number of users for each page;
- searchUser: the string by which the users name and email would be filtered by.

The controller then called the service responsible for fetching all the users with the corresponding teams, as seen below in Listing 2. It starts by doing a request to get all users, with the corresponding filters. After that, it counts the total number of users and calculates whether there is a next page by comparing the current page with the total count of users. Then it creates a new array "mappedUsers" with each user and an empty "teamsNames" array. It then fetches users with their corresponding teams and creates a set of unique user IDs from the fetched data. The final "results" object combines users with their teams and the users without teams (from "mappedUsers") using spread operators. The combined array is sorted alphabetically by first name and, if first names are the same, then by last name. Finally, the function returns the results object containing user data, pagination information, and team names.

```
async getAllUsersWithTeams(page = 0, size = 15, searchUser?: string) {
        const users = await this.getAllUsersWithPagination(page, size,
searchUser);

        const count = await this.userRepository.countDocuments();
        const hasNextPage = page + 1 < Math.ceil(count / size);

        const mappedUsers: UserWithTeams[] = users.map((userFound) => {
            return {
                user: userFound,
                teamsNames: []
            };
```

```
        });
        const usersOnlyWithTeams = await
this.getTeamService.getUsersOnlyWithTeams(users);

        const ids = new Set(usersOnlyWithTeams.map((userWithTeams) =>
String(userWithTeams.user._id)));

        const results = {
            userWithTeams: [
                ...usersOnlyWithTeams,
                ...mappedUsers.filter((user) =>
!ids.has(String(user.user._id)))
            ],
            hasNextPage,
            page
        };

        results.userWithTeams.sort((a, b) => {
            if (a.user.firstName === b.user.firstName) {
                return a.user.lastName < a.user.lastName ? -1 : 1;
            }

            return a.user.firstName < b.user.firstName ? -1 : 1;
        });

        return results;
    }
```

Listing 2 - Service for fetching users with teams (SPLIT)

5.4.2. Edit User

The second feature I developed was the edit page for each user. This page, as opposed to the previous one I described, only has access granted to users with the Super Admin role. This means that this page always had the same layout and features. Taking a look at Figure 20, it is clear that the layout has a lot of similarities with the users list page (Figure 19), such as: the breadcrumbs on top of the page, the title which in this case is the name of the user being edited, and the list of cards. The goal of this page was to show the list of teams a certain user belonged to. Each card corresponded to a team, displaying information about:

- Name: the name of the team. When clicked, it would redirect to the page of that team;

- Members: all team members. If the user hovered this section, the full list of members would show up;
- Team admin: the admins of the team, with the same hover feature as the members;
- Role: displayed the current role of the user on that specific team. The Super Admin had the ability to edit the role by clicking on it, which would show a list of all roles;
- Bin icon: when clicked, it would remove the user from that specific team.

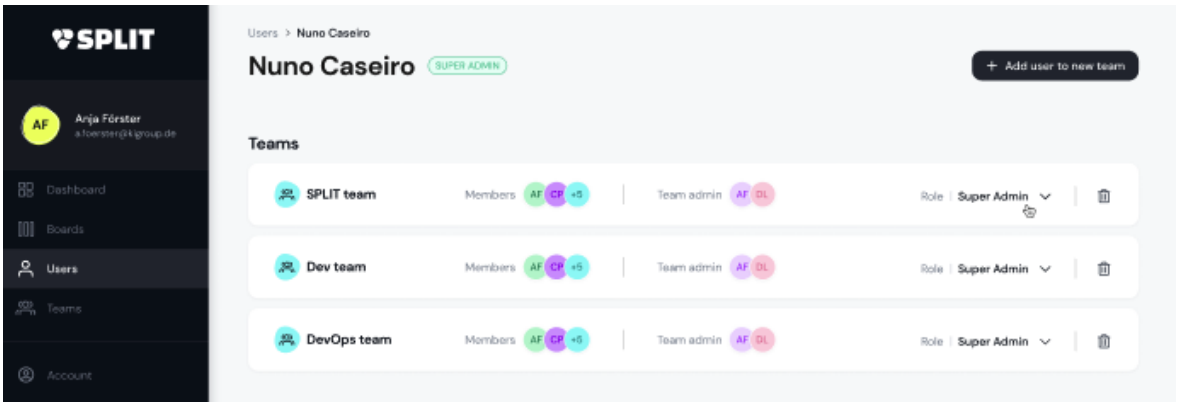

Figure 20 - Edit user page (SPLIT)

One last functionality this page had was the “Add user to new team” button. By clicking on it, a modal component would appear in the screen, as seen below in Figure 21. In this case, the teams being displayed were the ones that the user was not part of. The teams could be filtered by name, and after selecting one or more teams, the user was required to click on the “Save” button. In my opinion, the “Save” button is not user friendly because during one retro I got to be a team responsible and during the meeting with the Super Admin, he often forgot to click on the “Save” button, because he thought that only selecting the teams would be enough. With that said, I would change it to dynamically make an API call every time a team was selected. It could be seen as a trade-off, because that way there would be a lot more API calls, but since there was only one Super Admin in the company, the impact would be unnoticeable, making it a worthy improvement.

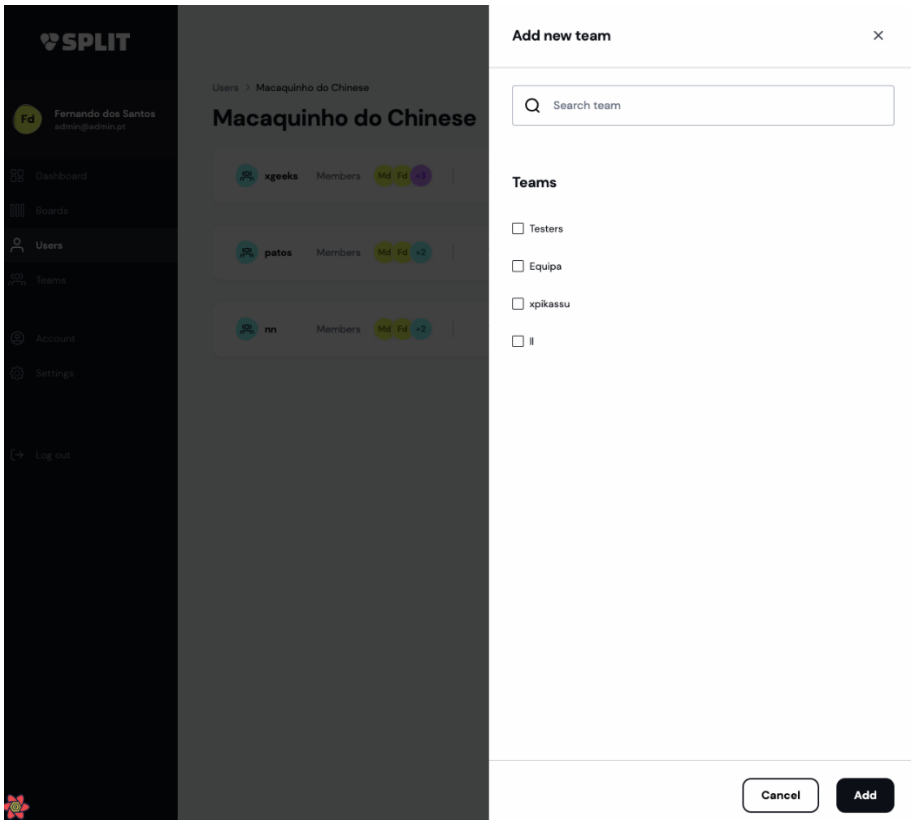

Figure 21 - Add user to a new team (SPLIT)

5.5.Summary

After the onboarding and training phase were completed, I joined a project for the first time, SPLIT. I was assigned the role of a Fullstack developer during my stay at the project, from 14th of November until 6th of January. SPLIT is an open-source retrospective tool developed by xgeeks, already at its later stages of development when I joined the team, which had its first release and use by the company a few weeks after I left the project.

Before SPLIT, xgeeks used EasyRetro as a retrospective tool, but it posed challenges for larger teams like manual member addition to a Slack channel, random team generation, and board maintenance. Comparing alternatives like TeamRetro and Retro.io, SPLIT emerged as the superior free option, enabling large teams to be split into sub-teams with sub-boards for easier management. Its open-source nature aligned with xgeeks' customization needs, setting it apart despite other tools offering more features in their paid versions.

This project embraced Agile with a Scrum focus, facilitating adaptability, value delivery, and teamwork. GitHub centralized user story and task management, breaking them

into tickets for clear progress tracking. Daily stand-ups synced the team, bi-weekly meetings gathered feedback and planned future steps. Pair programming with a fellow intern promoted knowledge sharing, efficient problem-solving, and collaborative success over the six-week project.

The architecture of this project was based on Next.js as the frontend framework and NestJS as the backend framework. It used a non-relational document oriented database, MongoDB. It had an integration with the Slack API to create/delete channels, as well as send messages. Redis was employed as a queueing system to manage and schedule jobs such as sending Slack messages and handling database queries. Another integration the application had was with the Microsoft SSO service, in order to register and authenticate users.

I was assigned the responsibility of developing two features: the page with the list of users and the page to edit users. Both of those tasks were successfully accomplished, and since I completed them faster than expected, I even had time to do bug fixes, which were very much needed because the application was scheduled to have its first release by the end of January.

6. Project 2 – Command Center

Upon completing my involvement in the SPLIT project, I was allocated to a new project, Command Center (also known as CMD Center), from 9th of January until 5th of May. Some senior engineers at xgeeks had the idea to create an internal tool, called CMD Center, that gathers multiple useful internal applications in one place. The first application to be developed in Command Center was the achievement application, also called "xAchievements" or Achievement System. The purpose of this application was to track everything that happened inside xgeeks and map it into achievements and give visibility to every engineer of what could be accomplished. These achievements ranged from a set of different categories, for example hiring, projects, GitHub/Gitlab milestones, training, and social events. Every engineer would have their profile, and as they completed achievements, they always received points, and in some cases physical or digital rewards. At a later stage, these points gained by completing achievements would be used in another application of the Command Center, which had the name "xStore", an application where users had the ability to exchange points for real life amusement experiences, physical items, and digital products.

6.1. Methodology

The methodology used in the development of this project had some differences in comparison to SPLIT. First of all, this time there was no pair programming. Secondly, besides using Agile with Scrum, we also used Kanban, which means the development methodology was a mix of both, also known as Scrumban. Kanban focuses on the visual representation of the development process, through a board. The board we used was the one provided by GitLab, where the tickets that were created had a label. In our case those labels were "To Do", "In Progress" and "In Review". Once reviewed, the tickets would be archived into a "Closed" section. The tasks (also named tickets or issues) to be done were mostly never pre-assigned to people. Instead, we followed one of the principles of Kanban, which is the "pull" concept, where the developers would have the freedom to assign themselves to tasks of the backlog, instead of a certain individual assigning tasks to people.

In terms of meetings, initially there were only three per week, on Monday, Wednesday, and Friday. However, a month later, those meetings became daily, at 10:30 in the morning. This was a decision made by the Product Owner, in an attempt to increase the probability of

the stakeholders being available to attend the meetings and provide direct feedback to the entire team.

6.2.Architecture

Just like the SPLIT project, CMD Center had the frontend built on top of Next.js and the backend built with NestJS, as seen in Figure 22.

Redis was utilized as a message broker. It kept track of the jobs that were not processed yet, such as external API calls and database queries. Speaking of API calls, the backend had an integration with the Slack API, in order to send a message to the user's Slack account each time an achievement was completed.

The only authentication method for this application was SSO, through Microsoft. Then in the backend, every time a user authenticated, it was possible to have access to multiple user attributes, such as his name, email, and photo, which would then be stored as a user record in the PostgreSQL database.

The main difference between the architecture of this project and the previous one is the database. As opposed to SPLIT, which used a non-relational document database (MongoDB), CMD Center used PostgreSQL, a relational database.

The application was deployed in Cloud Native, meaning it did not rely on storing persistent data within the application server itself (in other words, the app was stateless). Instead, images were stored externally using an Amazon S3 (Simple Storage Service) Bucket. In a simple way, a S3 Bucket is a file system used to store files. The way it was integrated in the application was as follows:

1. An image would be uploaded from the Web App to the API;
2. The image then was uploaded to the storage service (S3 Bucket) and retrieved to the API an identifier URL. That URL corresponded to the image;
3. That URL was stored in the PostgreSQL database table.

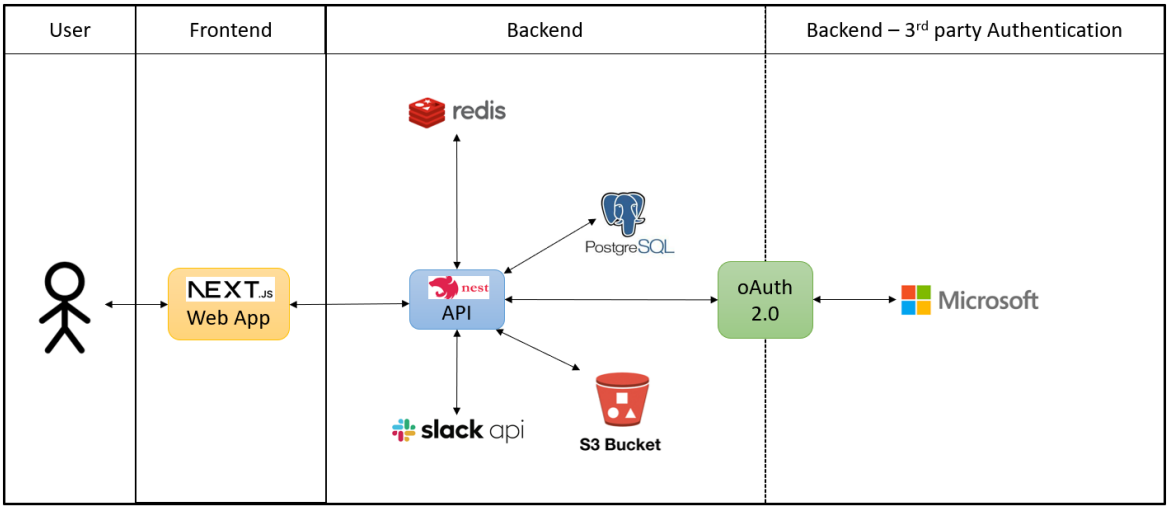

Figure 22 - CMD Center Applicational Architecture

6.3.Development

A core functionality of the xAchievements application relied on the users inputting their GitHub, GitLab and Greenhouse handles, because that way we were able to track metrics and automatically assign achievements that relied on those metrics. The application was integrated with GitHub, GitLab and Greenhouse by using webhooks, which were automated notifications that are triggered by specific events. In a version control system like GitHub and Gitlab, we were checking for events such as the creation of a new commit, pull request, issues, or merges. Greenhouse was an important tool used by the company in order to manage the hiring process in activities such as job postings, candidate resumes and interview scheduling. The webhooks in this case were used to track activities such as number of interviews a user has participated in, code challenges reviewed and created.

I was responsible for creating both the frontend and backend of the “Settings” page on the application (Figure 23). The objective of this page is for the user to insert its handles for GitHub and GitLab (at the time of the report, the Greenhouse integration was not done) in order for the application to create the integration and track user metrics needed for the achievements. As a “safety” mechanism, the submit button was disabled when the input was not dirty, which could happen in two scenarios: the first when the user entered the page and the input was empty, the second is when the user submitted the handle and made no further changes.

Before jumping into the implementation and the code of this feature, first I want to point out one concern that is related not only to this feature, but the application as a whole, which

is the reliability and traceability of user's activities. In some cases, there could be a dilemma for the developers who used their personal GitHub or GitLab account, instead of one created using the company's email, because otherwise it would be easy to track every account the same way, as the only criteria would be accounts with the domain "kigroup.de". In order to overcome that, we used 2 different solutions for each version control system:

- GitHub: repositories were tracked at the group level, the group being xgeeks;
- GitLab: since the option to track repositories at the group level was not free, like in GitHub, the only option was to set up individual webhooks for each repository within the group. If xgeeks had the GitLab Premium or Ultimate versions, a single group-level webhook would be enough, making it much easier to manage.

Another key aspect to point out in regard to limitation in tracking developers' activities is the fact that xgeeks, being a consulting company, had the majority of its projects for outside clients. In practice, it means that the xAchievements application was limited to tracking mostly internal projects as those were the ones that were part of the xgeeks repositories in GitHub/GitLab. In my opinion, this is the application's biggest shortcoming, as it affects the reliability in terms of developers' activities, because most people worked for external clients, so their work there wouldn't be tracked, making it way harder for them to complete achievements when compared to other employees who are working fulltime on internal projects.

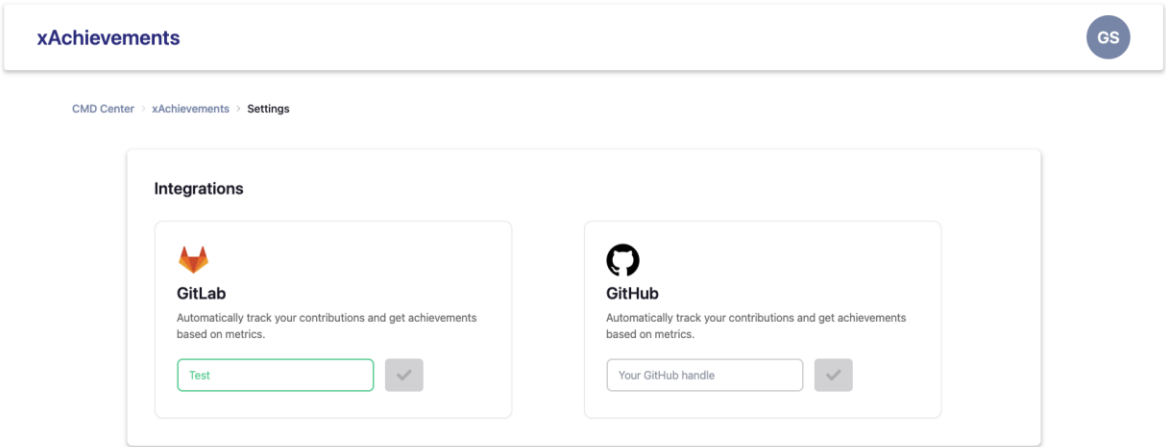

Figure 23 - Settings page (xAchievements)

Listing 3 corresponds to the code of the settings page. It starts with the “useTranslation” hook, which is a function provided by the “i18next” library [77] that enables translation support in a React component. In this case, the hook goes to the settings file, which is in the “/public/local/en/” directory (configured in the “next-i18next.config.js” file) and by calling the “t” function returned by this hook, it’s possible to easily access translated strings for the app based on the user's selected language. This function takes a string key as an argument and returns the corresponding translation for that key. After the translation part in the code, there is a state variable for the user, which is fetched through the “useQuery” method every time the page is loaded. This way, we make sure the handles are always up to date, which is important. The Wrapper, Head and Breadcrumbs components are used on every page file in the application, so it makes sense to quickly explain both, even though they have a fairly simple implementation:

- Wrapper: It simply renders a div element with a specific style and some padding and margin styles and wraps around its children. It is designed to be used as a parent component that wraps around the main content of each page in the application, with the goal of providing consistent styling and layout across all pages;
- Head: it is a built-in component in Next.js that allows to modify the head element of a page and add metadata. In this application it was only used to modify the title element, as the other elements and metadata were not important because since it was an internal application to xgeeks, there was no need for SEO optimization;
- Breadcrumbs: it is a navigation aid to help users understand their location within a website’s hierarchy. Looking at Figure 23, it corresponds to the upper part where it says “CMD Center -> xAchievements -> Settings”. If the user clicks on any of those, the web application will redirect to the corresponding page, through the use of “useRouter” hook (Next.js native hook) and its push method.

The Text component corresponds to a primitive component, which creates a span element with the content inside, in this case it is the “Integration” text above the cards in Figure 23. Finally, there are 2 components named “GitLabCard” and “GitHubCard”, which correspond to both providers.

It can also be seen in the div element that it uses the “className” attribute, which belongs to the CSS framework Tailwind. This framework was used to shape the design of the entire application.

```
const SettingsPage: NextPage = () => {
  const { t } = useTranslation('settings');
  const [user, setUser] = useState<User>();

  useQuery<User, ErrorWithCause>(['user'], () => userService.fetchUser(),
{
    onSuccess: (userData) => {
      setUser(userData);
    },
  });

  if (!user) return null;

  return (
    <Wrapper>
      <Head>
        <title>{t('pageTitle')}</title>
      </Head>
      <Breadcrumbs />
      <div className="shadow-default mx-auto p-10 mt-10 rounded-lg">
        <Text customStyle="font-semibold text-2xl"
content={`${t('integrations')}`} />
        <div className="grid mt-8 gap-5 grid-cols-1 lg:grid-cols-3">
          <GitLabCard user={user} />
          <GitHubCard user={user} />
        </div>
      </div>
    </Wrapper>
  );
};

export const getServerSideProps: GetServerSideProps =
withAuthorization(async ({ locale }) => ({
  props: {
    ...(await serverSideTranslations(locale || '', ['common',
'settings'])),
  },
}));

export default SettingsPage;
```

Listing 3 - Settings page file (xAchievements)

Both card components of GitHub and GitLab are similar, with the only difference being the props passed to the IntegrationCard component, which is the component with the majority of the functionalities of this Settings feature. It is explained in detail in Appendix A. With that said, let's take the GitLabCard as an example, as seen in Listing 4. This component is quite simple, as it only returns an IntegrationCard component. The image and title props are a simple string, the provider is a "Gitlab" string from an enumeration, and then there is the "formSchema", which is more complex, so it will be explained in the next paragraph. The description and placeholder both take in a translation function that returns the associated string in the "settings" file. Note that the placeholder translation function also takes in an object with a provider property, meaning that the translation file has a string with a variable called "provider" which will be replaced with "Gitlab" in this case. The handle prop takes an expression that will return the user's GitLab handle in case it exists, otherwise it will return an empty string. The last prop, the "data-testid" simply takes in the string that will identify the whole component for testing purposes.

```
import IntegrationCard from
'@/components/shared/IntegrationCard/IntegrationCard';
import { getGitlabFormSchema } from '@/constants/integrations';
import { ProviderEnum } from '@/types/settings';
import { User } from '@/types/user';
import { useTranslation } from 'next-i18next';

const GitLabCard = ({ user }: GitLabCardProps) => {
  const { t } = useTranslation('settings');

  return (
    <IntegrationCard
      image="/gitlab-logo.png"
      title="GitLab"
      provider={ProviderEnum.Gitlab}
      formSchema={getGitlabFormSchema(t)}
      description={t('integrationDescription')}
      placeholder={t('integrationHandleInputPlaceholder', { provider:
'GitLab' })}
      handle={user?.handles?.gitlab ?? ''}
      data-testid="gitlab-integration-card"
    />
  );
};

export type GitLabCardProps = {
  user: User;
};
```

```
export default GitLabCard;
```

Listing 4 - GitLabCard component (xAchievements)

In order to check if the handle of the user inputs is valid, a schema validation is passed to the “formSchema” prop. This schema is created by using the yup library [78], which is a schema builder for runtime value parsing and validation. Below in Listing 5 are both schemas for validating GitHub and GitLab handles (username). These rules were taken from the create an account page of both providers, which list all the rules that must be obeyed to create a valid username. Since the code is similar in both scenarios, let’s explain the GitLab schema validation. The “getGitlabFormSchema” function takes in a translation function that is used to show an error message in case any validation fails and returns a yup object schema validation, which will later be used to validate the form. Then the “yup.object().shape()” method is used to define the validation schema for the GitLab form. In this case, it requires the form to have a 'handle' field that must be a string between 2 to 255 characters long using the “.min()” and “.max()” methods, where the second parameter is a string returned by the translation function, that serves as error message in case the validation condition does not apply. The “yup.string()” method specifies that the field must be a string. Finally, the “.matches()” method is used to ensure that the 'handle' field only contains alphanumeric characters and excludes empty strings from being validated. It takes in a regular expression as an argument, which matches any alphanumeric character using the “^[a-zA-Z0-9]*$” pattern. If the 'handle' field contains any other characters, an error message will be generated using the translation method.

```
import { TFunction } from 'next-i18next';
import * as yup from 'yup';

export const getGithubFormSchema = (t: TFunction) =>
  yup.object().shape({
    handle: yup
      .string()
      .min(1, t('minChar', { number: 1 }))
      .max(39, t('maxChar', { number: 39 }))
      .matches(/^[a-zA-Z0-9-]*$/, {
        message: t('alphanumericOrHyphens'),
        excludeEmptyString: true,
      })
      .matches(/^((?!-{2,}).)*$/, {
```

```
        message: t('cantContainConsecutiveHyphens'),
        excludeEmptyString: true,
      }),
  });

export const getGitlabFormSchema = (t: TFunction) =>
  yup.object().shape({
    handle: yup
      .string()
      .min(2, t('minChar', { number: 2 }))
      .max(255, t('maxChar', { number: 255 }))
      .matches(/^[a-zA-Z0-9]*$/, {
        message: t('alphanumeric'),
        excludeEmptyString: true,
      }),
  });
```

Listing 5 - Schema validation file (xAchievements)

Testing is an essential part of software development as it helps ensure that the software meets the desired requirements and functions correctly. In this project we performed unit tests on both the frontend and backend of the application, in order to make sure all components and functions behave as expected every time there is new code merged. For this feature (settings page), the main components are the GitLabCard, GitHubCard and IntegrationCard. Both the GitHub and GitLab cards have very similar code, since all they do is call the IntegrationCard with different properties, so the unit tests done for both components are the same. Below in Listing 6 is the test file for the GitLabCard component, which tests the successful render of the component. The test imports the GitLabCard component, a user factory to create a test user, and a custom rendering function from a testing utility module. The render function is called within a test block that checks if the component is successfully rendered by verifying that a specific test ID is present in the rendered component's DOM.

```
import GitLabCard from '@/components/GitLabCard/GitLabCard';
import UserFactory from '@/utils/factories/user';
import { renderWithProviders } from '@/utils/testing';

const user = UserFactory.create();

const render = () => renderWithProviders(<GitLabCard user={user} />);
```

```
describe('GitLabCard', () => {
  it('should render successfully', () => {
    expect.assertions(1);

    const { getByTestId } = render();

    expect(getByTestId('gitlab-integration-card')).toBeInTheDocument();
  });
});
```

Listing 6 - GitLab component test file (xAchievements)

The IntegrationCard component is a bigger component in terms of code and logic, so it has, consequently, more test cases. The unit tests for that component can be seen in Appendix B.

When it comes to the backend of this feature, the webhooks were assigned to another developer. In practice, there was a table in the database called “oauth” that was used to register all the handles that the users have in certain providers (GitHub, GitLab and Greenhouse). Also, at the time I started working on the backend of this feature, there was already an endpoint that handled the POST request to register the handle of a user, so I was responsible for creating the PUT endpoint to update the handle of a user. One important thing to point out is that after I finished the update endpoint and created the merge request for this feature, I was suggested by the reviewer (the developer who created the POST endpoint for this feature) to refactor the code and instead of having his POST endpoint and my PUT endpoint, to join them both into a single PATCH endpoint, because the “save” method (called on an instance of the Oauth repository) we were using on both endpoints works as follows: it persists an entity to the database, creating it in case it doesn’t exist or updating it if it there is already a record of it. However, as I’ve learned throughout the bachelor and masters, a PATCH endpoint should be used when updating a resource with partial modifications. With that in mind, although it could work on practice to have the endpoint as a PATCH, from a theoretical point of view it did not make sense to create a new register in the database through a PATCH endpoint, so I suggested using a POST instead, and after discussing it the reviewer agreed with my idea.

The controller method for this endpoint ended up being like in Listing 7. The controller handles a POST request to the endpoint “/api/v1/auth/integrations”, because the backend of

the application has the default access endpoint of "/api/v1", and the "auth" in the endpoint corresponds to the domain of the controller that the "/integrations" endpoint belongs to. It is protected by a "JwtAuthGuard" (only authenticated users can access the endpoint with this guard) and expects two parameters: a "requestUser" object representing the authenticated user making the request, and a "CreateUpdateIntegrationHandleDTO" object containing the user handle and the integration provider to be associated with the user. The method then calls a service method responsible for the logic behind this endpoint.

```
@Post('integrations')
  @ApiOperation({ summary: 'Associate integration handle to user' })
  @UseGuards(JwtAuthGuard)
  updateOrCreateIntegrationHandle(
    @RequestUser() requestUser: User,
    @Body() updateIntegrationHandleDTO: CreateUpdateIntegrationHandleDTO,
  ) {
    return this.authService.updateOrCreateIntegrationHandle(
      requestUser,
      updateIntegrationHandleDTO,
    );
  }
```

Listing 7 - Integrations controller to create/update (xAchievements)

Listing 8 below shows the code of the service that creates or updates an integration. It starts by destructuring the external handle and the provider from the DTO (Data Transfer Object), and then checking if the provider exists, by using the "findOneBy" method in the AuthProviders repository. In case the provider is not registered in the database, it throws an exception, which the user will be able to see in a pop-up in the frontend web page. The Auth Provider id is extracted, because it is necessary for the next queries since there's a column in the Oauth table for it, in order to associate the provider to that register. After getting the id of the provider, two queries to the Oauth repository are made, with the goal of extracting two different variables with different meaning:

- oldAuth: this variable stores the result of the first query, which returns the Oauth register of the user that made the request to the controller of this service for a given provider, also referenced by the user who made the request. If the query returns an Oauth object, it means the user already has a handle for the given

provider, so it should be updated. In case the variable is undefined, it means the user doesn't already have the integration done, so it will be created;

- newAuth: the second query searches in the database if there already exists a user with the same handle-provider pair, because it cannot be possible. For example, in GitHub (or any other service/application), user handles must be unique, so when a user tries to create an account with a handle that already exists, GitHub would give error in that instance. The goal is for this variable to be empty, but if it isn't, there are two different scenarios that can occur. First, if the Oauth found has the same id of the user that made the request, it means the user has submitted a handle equal to the one he already has, so it simple returns the object (in my opinion, it would make more sense to throw an exception, because the handle is invalid since it already exists, but it was decided this way by the reviewer). The second scenario is when a different user than the one who made the request already has the submitted handle in the given provider, resulting in an exception being thrown and the error message will be seen by the user in the web page).

Finally, if the happy path (input and output are as expected) was followed, the service comes to an end by creating or updating the Oauth repository using the save method, which takes in the object to create/update. In this case, a "Object.assign()" method is being passed in order to be used to copy the values of all objects it receives in the parameters into a new object. The way it works is as follows: in case the "oldOauth" object exists, it will merge with the second object, which in practice will only update the handle; in case an "oldOauth" doesn't exist, a completely new object is created and saved into the database. The resulting updated/created object is returned as response.

```
async updateOrCreateIntegrationHandle(
    requestUser: User,
    updateIntegrationHandleDTO: CreateUpdateIntegrationHandleDTO,
  ) {
    const { externalUserHandle, provider } = updateIntegrationHandleDTO;

    const authProvider = await this.authProviderRepository.findOneBy({
      name: provider,
    });

    if (!authProvider) throw new AuthProviderNotFoundException();

    const { id: providerId } = authProvider;
```

```
    const [oldOauth, newOauth] = await Promise.all([
      this.oauthRepository.findOneBy({ providerId, userId: requestUser.id
}),
      this.oauthRepository.findOne({
        relations: { user: true },
        where: { providerId, externalUserHandle },
      }),
    ]);

    if (isNotEmptyObject(newOauth)) {
      if (newOauth.userId === requestUser.id) {
        return newOauth;
      }

      throw new BadRequestException(
        'Another user with the same provider already has the same handle',
      );
    }

    return this.oauthRepository.save(
      Object.assign(oldOauth || {}, {
        userId: requestUser.id,
        providerId,
        externalUserHandle,
      }),
    );
  }
```

Listing 8 - Integrations service to create/update (xAchievements)

To ensure the quality and correctness of this code, NestJS provides a built-in testing framework [79] that allows developers to write unit, integration, and end-to-end tests (unit tests only in this application). These tests are written using the testing library Jest and are run with NestJS's test runner. In order to test this service (Listing 8), I made in total 5 different iterations, for 3 different scenarios:

- When the AuthProvider exists: should create Oauth integration; should update a user handle;
- When AuthProvider does not exist: should throw a “AuthProviderNotFound” exception;
- When the handle already exists on the given provider: should throw a “BadRequestException”.

Since there are many iterations, in order to avoid repetition/redundancy I will just show one “happy path” scenario and “sad path” scenario (when something goes wrong).

Starting by the happy path scenario, in Listing 9, the name indicates this test is for a scenario when the user is creating a new Oauth integration for a valid AuthProvider. This iteration starts with the “useSeeding” asynchronous function to setup the test data, which is always used when a Factory is used to generate data. Factories are functions that create instances of classes or objects with predefined properties, in this context they are objects with the same properties as in the database, to simulate a database register. Then, the User and AuthProvider factories are called to create instances of these entities. After that, the “mockAuthProviderRepository” is mocked to return the AuthProvider instance created by the factory (in other words, it is simulating that the database has that register). The “updateOrCreateIntegrationHandle” method is then called with the user instance and an object containing the “authProvider” name and a random external handle. Finally, an assertion is made that the “mockOauthRepository.save” method is called with the correct arguments, which indicates that the Oauth integration has been successfully created.

```
describe('when authProvider exists', () => {
      it('should create oauth integration', async () => {
        await useSeeding();

        const [user, authProvider] = await Promise.all([
          factory(User)().make(),
          factory(AuthProvider)().make(),
        ]);

        mockAuthProviderRepository.findOneBy.mockResolvedValueOnce(
          authProvider,
        );

        await authService.updateOrCreateIntegrationHandle(user, {
          provider: authProvider.name as AuthProviderEnum,
          externalUserHandle: 'test',
        });

        expect(mockOauthRepository.save).toHaveBeenCalledWith({
          providerId: authProvider.id,
          externalUserHandle: 'test',
          userId: user.id,
        });
      });
```

Listing 9 - Integration test for happy path (xAchievements)

Regarding the other scenario when something goes wrong, Listing 10 depicts a scenario where the AuthProvider exists but the handle that is submitted is the same as another user. As stated in the previous scenario, all tests' iterations start with the setup of test data, and in this case, it also expects two assertions to be true, otherwise the test will fail. Initially, three variables are made, the first two correspond to the user that is making the request and the user that already exists, and the third variable is the provider for this test, which needs to be the same for both users in order to test the logic of this scenario. A fourth variable is then created with the aim of representing the existing Oauth in the database, which is then passed to the mock Oauth repository. One important thing to point out is that it was not necessary to create a mock for the Provider and the User. The reason for that is because the TypeORM cascade relations in the Oauth allows for automatic creation of those database records in case they do not exist yet. That's the "power" of using cascade in entities' properties. The final part of the test is a try-catch block, where it tries to call the service with a new user and the same provider and handle as the Oauth register that was mocked. The result of this is the code will run in the service (Listing 8) and a "BadRequestException" will be thrown, stopping the database from creating the new Oauth register, so the save method on the service will not be called, which is the expected, in Listing 10.

```
describe('when handle already exists on that provider', () => {
      it('on another user', async () => {
        await useSeeding();
        expect.assertions(2);

        const [user, user2, provider] = await Promise.all([
          factory(User)().make({ id: 1 }),
          factory(User)().make({ id: 2 }),
          factory(AuthProvider)().make({ name: 'github' }),
        ]);

        const oauth = await factory(Oauth)().make({
          user,
          provider,
        });

        mockAuthProviderRepository.findOneBy.mockResolvedValueOnce(provide
r);
        mockOauthRepository.findOne.mockResolvedValueOnce(oauth);

        try {
          await authService.updateOrCreateIntegrationHandle(user2, {
            provider: provider.name as AuthProviderEnum,
```

```
            externalUserHandle: oauth.externalUserHandle,
          });
        } catch (ex) {
          expect(ex).toBeInstanceOf(BadRequestException);
          expect(mockOauthRepository.save).not.toHaveBeenCalled();

        }
```

Listing 10 - Integration test for sad path (xAchievements)

When I first joined this project, although the tech stack was similar to SPLIT, I noticed the structure of the frontend was greatly lacking in Primitive components. One of the main reasons for it was the fact the SPLIT had full app design made by professional designers, as opposed to xAchievements, where the design was provisional and only made by the developers. The reason the design had an impact on this matter, is because Primitive components are reusable blocks across the application, which avoid code duplication and allow for consistency in both functionality and design. Two months after I joined this project, we were given the information that having an available designer to work on the project was not in the cards for a couple of months, at least. With that said, a member of the team took my initial advice of increasing the number of Primitive components in the application, and since the vast majority of the features were finished, as well as bug bounty, the team decided to create more Primitive components.

Below in Figure 24 is the folder containing all the Primitives of the project. Initially, when I joined the project there were five: Avatar, Breadcrumbs, Card, Text, and Wrapper. Just before leaving the project, there are 12, out of which some replace native HTML elements, like the Button, Checkbox, Form, and Input, while the others are custom elements Primitives. The explanation of the Button primitive is given in Appendix C.

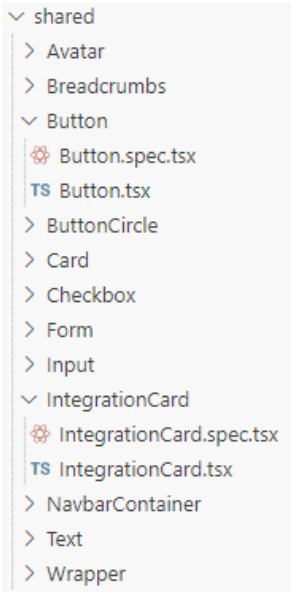

Figure 24 - Primitives folder (xAchievements)

Besides the settings page, which was the main feature I developed, I also worked on other features, bug fixes and increased the number of unit tests. Appendix D shows some more features I developed in this project.

6.4.Summary

I worked as a Fullstack developer (with more focus on the frontend) from 9th of January until 5th of May in the Command Center project. It was designed to be a hub for internal applications, xAchievements being the first application and the one I was part of. xAchievements was an application with a list of achievements responsible for tracking information about the activity of xgeeks' engineers such as GitHub/GitLab activity, attendance to social events, referrals and hiring interviews. The goal was for the engineers to complete these achievements and in return receive points, which at a later stage would be redeemable for gifts.

The development methodology consisted of a hybrid of Agile with Scrum and Kanban (Scrumban), utilizing a GitLab board for visual progress tracking, and embracing the "pull" concept for task assignment. Initially, meetings occurred thrice weekly but shifted to daily sessions later to accommodate stakeholder feedback.

In terms of architecture, this project was built with Next.js for the frontend. The backend was built with NestJS, and it had an integration with the Slack API in order to send messages

to users. It used a PostgreSQL database to store and retrieve data and S3 Bucket to store images. Redis was used as a queueing system for job scheduling. There was also an integration with the Microsoft login service in order to register and authenticate users in the application.

During the development of this project, I performed mainly frontend related tasks because there was a backend developer working full time on the project. Similarly, to the previous project, I joined at a later stage of development, meaning I did mostly bug fixes and increased test coverage. I also occasionally completed some minor features, with the exception of the settings page, which I built from scratch and required the most effort to complete.

7. Project 3 - Geekathon

The last project I was assigned was the Geekathon's website. I was part of the team from 8th of May to 3rd of July. This time, I was assigned the role of frontend developer, as opposed to the previous two projects where I performed tasks both on the frontend and backend of the applications.

Ever since 2021, xgeeks hosts a yearly event called "Geekathon", which is basically a hackathon during the whole weekend. This is an event where all people with legal age can participate and must be present in Leiria in the announced location during the event, with previous registration. To register, a person must buy a ticket through the website, with a varying price depending on the perks involved. There is always a challenge, which is a set of categories that are merely an indication of the path each team can take. The final product is a deliverable, which can be one of two formats:

1. Minimum Viable Product (MVP) version of a digital tool, such as a mobile or web application, website, tool, algorithm, or platform, that offers an innovative and original solution to the challenge;
2. Document that outlines the problem, solution, benefits, competitive landscape, and a business plan in a clear and illustrative manner. Visual representation of the solution is encouraged.

In the end, there is an evaluation by a jury. Judging criteria for the project include Business Potential, Impact, Innovation, and Pitch quality, with a focus on the ability to generate revenue, adoption by users, originality, and team performance.

The stack used in the frontend is, once again, Next.js and TypeScript, which was helpful because until then all projects I was part of also used those technologies for the frontend, so integration into this last project was quick and easy.

7.1. Methodology

The development of this project relied, once again, on Agile methodology, as it allowed us to respond to changing requirements and deliver high-quality software in a timely and efficient manner.

One of the key features of Agile is the use of short, frequent meetings called "dailies" or "stand-ups" to keep the team in sync. These meetings were held three times a week, during which team members discussed their progress, any issues they have encountered, and plans until the next meeting. The aim of these meetings was to ensure everyone was on the same page and identify any obstacles that needed to be addressed.

Another important aspect of Agile is the use of backlogs to manage the work, which is where Kanban comes in. In this project, similarly to CMD Center, we used GitLab boards as our backlog. This allowed us to prioritize and track the progress of tasks, ensuring that the team was always aware of what needed to be done.

Finally, we held weekly sync meetings with the product owner to ensure that we were meeting the expectations and delivering value. This meeting allowed us to get feedback, clarify requirements, and adjust the backlog if necessary.

7.2.Architecture

The application was built using Next.js as the frontend framework, just like all projects I was part of during the internship. The main difference is that the backend of the application was built with Laravel [80], a free and open-source PHP (Hypertext Preprocessor) framework. The architecture is illustrated in Figure 25.

In the frontend, the Web App had an integration with Unleash [81], a feature management service. Due to the fact the frontend was made using Next.js, I used a Next.js SDK in order to connect the application with the service. The purpose of this service was to enable/disable certain features for the end user. More information about this will be given in the next subchapter.

When it comes to the backend side of the application, although I was not responsible for developing any of it, that is where the majority of the services and integrations are. With that said, the next paragraphs will give an overview on what each component does.

Starting with the authentication, it was a crucial point of the application because it involved all the running parts. In order for a user to buy tickets, which is the main purpose of the application, the user needed to be authenticated. At the time I was in the project, the Web App only had the option for authentication with third-party services such as GitHub, GitLab, Google, Twitter, and LinkedIn. The user needed to click on one of those options,

which corresponded to an API URL that redirected to the corresponding service's authentication page. After filling in the login form, in case of error the provider itself would warn the user. In case of success, if the user had previously logged in the application, he would simply be redirected to the home page of the Web App. Otherwise, the provider would send back to the API a user object, resulting in the creation of four jobs, added to the Redis queue:

1. A welcome email using Mailgun;
2. Avatar upload to the S3 Bucket;
3. The creation of a new user in the user's database table (PostgreSQL);
4. A call to the Stripe API, using the Stripe Laravel SDK, to create a new Stripe customer account.

Mailgun [82] is an email automation service that provides APIs for sending, receiving, and tracking emails. During my stay at the project, there were only two instances where emails were being sent: when users first authenticated, they would receive a welcome email and when a ticket was purchased, an email with the receipt would be sent as well.

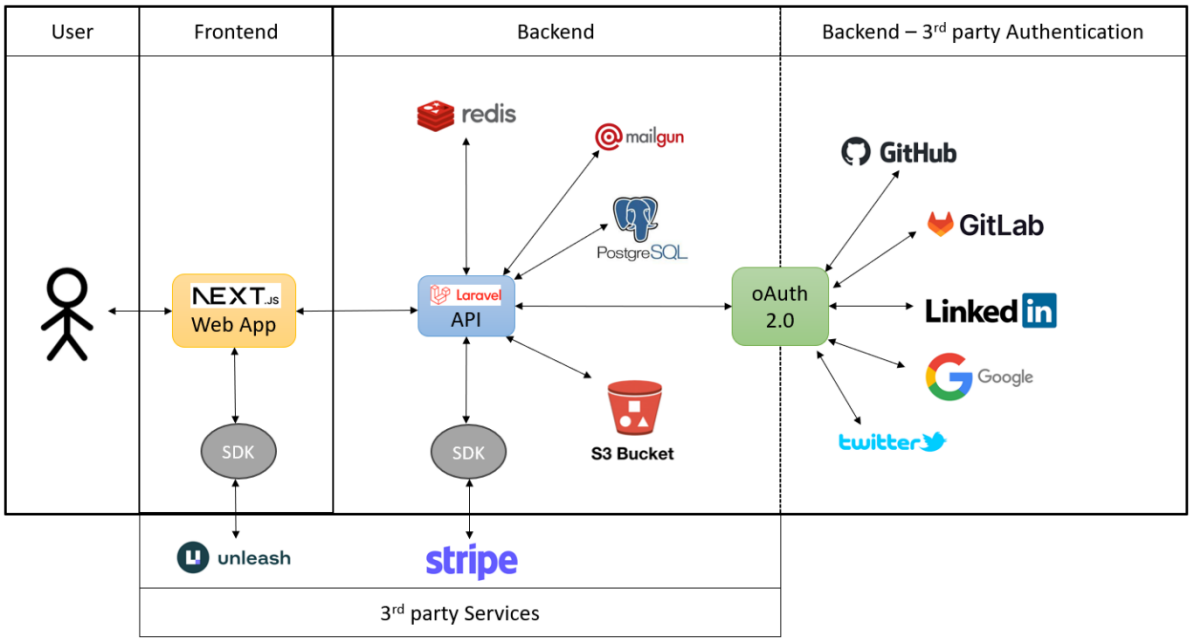

Figure 25 - Geekathon Applicational Architecture

7.3.Development

In this project, the responsibility of creating most of the issues fell on the tech leads. They had the expertise and understanding of the project requirements to identify and outline the tasks that needed to be developed. However, our team culture encouraged all members to contribute by creating issues whenever we felt the need for it. Personally, I took an active role in creating issues regularly. Being an autonomous individual with a strong understanding of the project goals, I was able to identify what needed to be done and effectively divide it into manageable sub-tasks.

As previously mentioned, a consistent practice across the three projects I worked on was the use of templates for creating these issues/tasks. These templates served as a guideline for structuring the information in a standardized manner. A typical issue follows a specific format, including various key elements:

- Title: resume of what needed to be done. Started with an identifier, depending if it was the frontend (GH-UI) or the backend (GH-CORE), followed by a few words describing the issue. For example, an issue for the frontend can have the following title: "[GH-UI]: mobile view for hero section";
- Description: contained two sections, for a brief discussion of the proposal and acceptance criteria;
- Assignee: the person who's responsible for that issue;
- Labels: group of words used to describe the issue. Some of the most commonly used were state labels ("TO DO", "IN PROGRESS", "IN REVIEW", "DONE") and description labels ("FRONTEND", "BACKEND", "DEVOPS", "FEATURE", "BUG").

Below in Figure 26 is a list of issues that needed to be developed at the time. The GitLab page only showed part of the information regarding each issue, like the title, who created it and the label. Furthermore, when clicking on an issue from the list, it was possible to see the description, which contained the crucial information on what needed to be done, as seen in Figure 27. In that figure the two major sections can clearly be seen, the Proposal (describing what needed to be done) and the Acceptance Criteria (all the criteria that needed to be obeyed in order for the code to be accepted and merged).

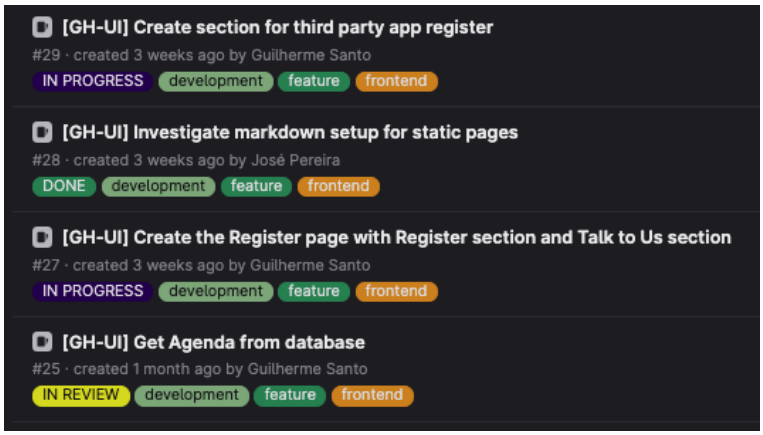

Figure 26 - List of issues (Geekathon)

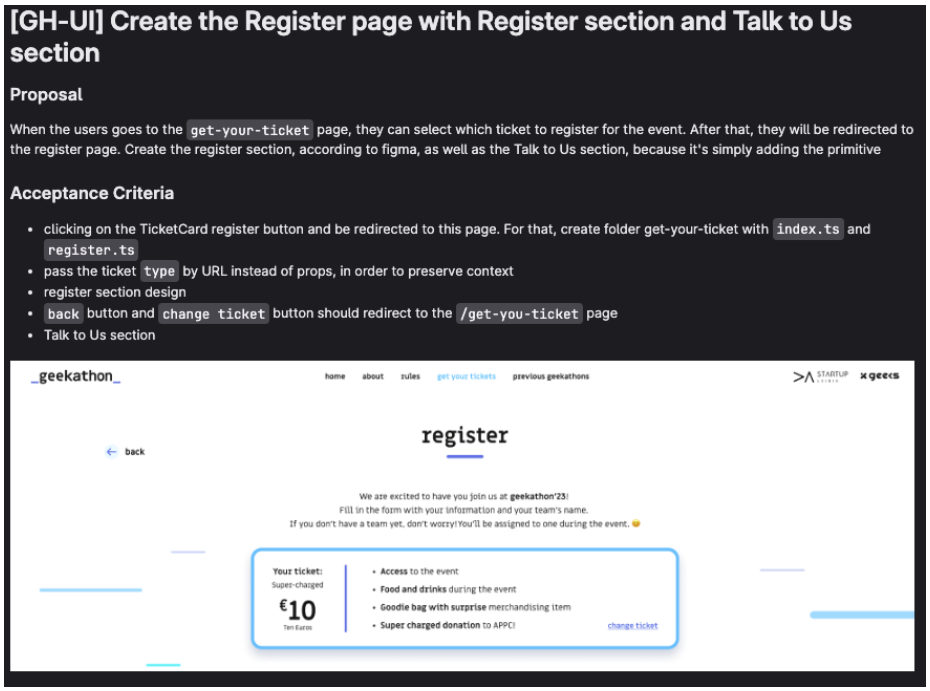

Figure 27 - Example of an issue (Geekathon)

During the first month, there was not a robust pipeline, meaning that there was only one main branch, called “main”. There were development and production environments, but the “main” branch was always used on all environments. About a month into the project, after the pipeline and infrastructure side of the project already had a good foundation, the team decided to discuss the fact we only had one main branch. The final decision was to have both a “dev” and “main” branches. The two main reasons for that decision were the following:

- Taking advantage of the version control system, GitLab, in order to have different branches for different lines of development. The “main” branch was used for stable production code, whereas the “dev” branch was used for ongoing development work;
- By having only one branch, we wouldn’t be able to do hotfixes without deploying the dev code to production. Hotfixes are urgent bug fixes or critical updates that need to be deployed quickly to address issues in the production environment. By having separate branches for development (dev) and production/stable code (main), we could isolate hotfixes from the ongoing development work.

The next paragraphs will reference and explain the features I fully developed from scratch. The design was implemented into the application by using the CSS framework Tailwind, just like the CMD Center project.

The first feature I was assigned to do when I joined the project was the Navbar (Figure 28). It consists of a parent div element, that contains three div elements inside:

1. Geekathon logo: A Link (Next.js Component that extends the HTML anchor tag) with an image of the logo inside. When clicked, it redirects the user to the home page;
2. Navigation links: a list of the main pages of the application, so that the user can easily navigate to them when clicked on;
3. Sponsor logos: Same logic as the Geekathon logo, except that when clicked, it redirects to the corresponding organizations’ website. Since it redirects the user to an external link, I took some security measures by using the anchor property rel="noopener noreferrer". The “noopener” prevents newly opened page from accessing the window.opener property, enhancing security. The “noreferrer” provides a level of privacy by preventing the destination page from knowing the exact URL of the page that referred the user.

Another functionality of the navbar is that it highlights, in blue, the page the user is currently in. As seen in the top of Figure 28, the user is on the home page (index file, according to the Next.js routing system, explained in the Next.js subchapter).

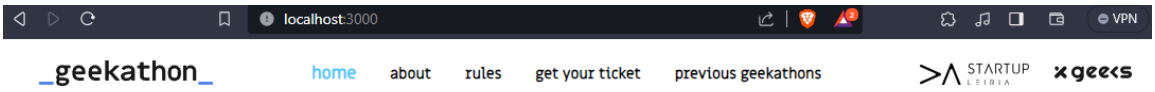

Figure 28 - Navbar (Geekathon)

In Figure 29 can be seen both the “talk to us” and the footer sections. I decided to show them together, due to the fact that a lot of pages had these two sections together in the end.

The “talk to us” section included three buttons for xgeeks' social media platforms: Twitter, Instagram, and LinkedIn. Below the buttons, there was a text informing users that they could contact xgeeks for any questions. The section also featured an email address highlighted in blue, as well as a "write us a message" button. Both the email address and the button were implemented using a Link component with a href attribute corresponding to, in this case, “mailto:hello@geekathon.dev”. When the href attribute has the “mailto” in the beginning of the string, it means that when the user clicks on the element, the browser would ask the user to open the default mail application, which would then have the “hello@geekathon.dev” as the destination.

The footer section was present at the end of all pages of the application, which is the expected behaviour of a footer. Due to some pages content not filling the entire page, I had to make sure the footer was sticky to the end of the viewport on those instances, because otherwise this section could show in the middle of the screen, which would look unprofessional. The footer itself consisted of three main div elements. The first one being a list of links to the home page, legal information pages and a FAQ (Frequently Asked Questions) page. The second div element simply contained a copyright issue, and the third had the social media buttons the user could click to be redirected to the corresponding social media.

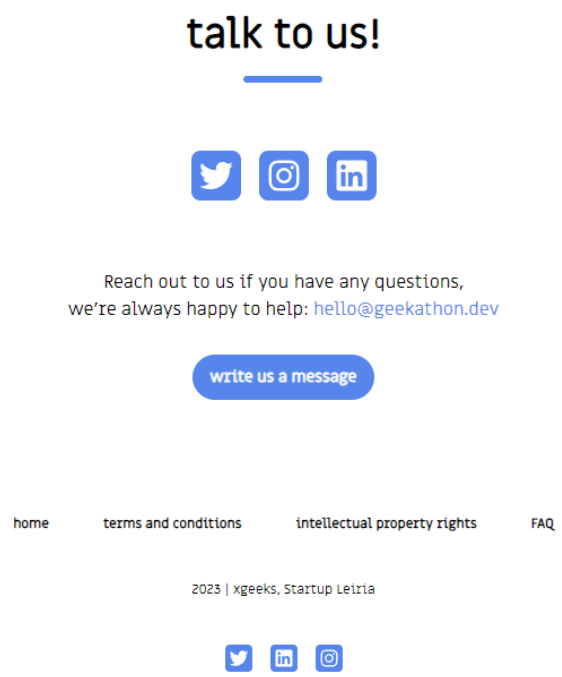

Figure 29 - Talk to Us section and Footer section (Geekathon)

Figure 30 illustrates what the agenda section looks like with dummy data (placeholder data used for testing or demonstrating a purpose). This section was part of the home page. Because the component for this section relied on static data (data that didn't change after render), I implemented the getStaticProps data fetching function in the home page. The data here, although from the database, was intended to always be the same, only minor adjustments could happen for example in the case of typos. With that said, using getStaticProps made sense here, making the application's main page load fast, in under a second (because the page was rendered on the server side, at build time).

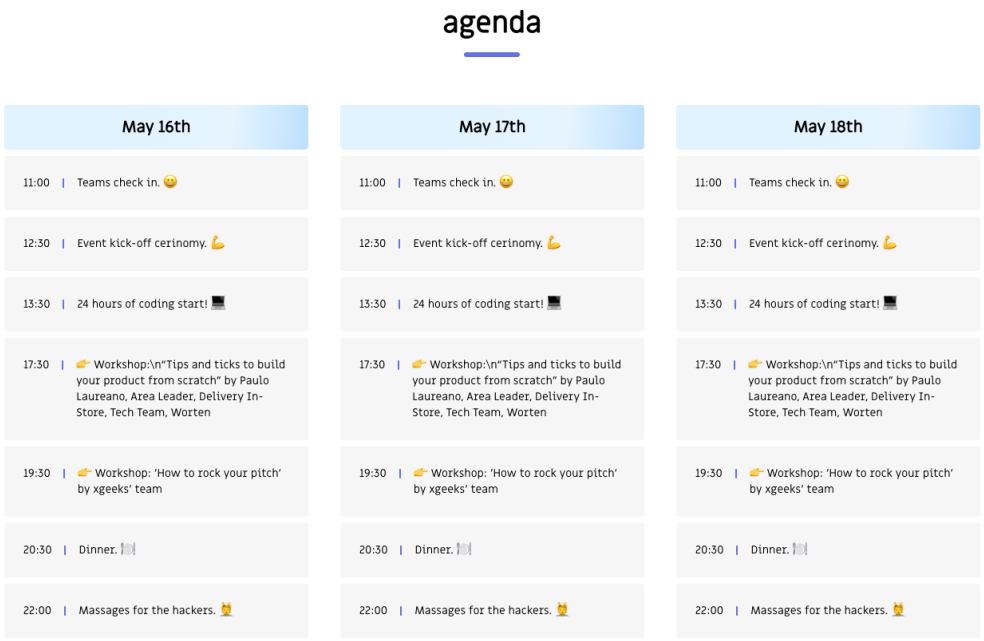

Figure 30 - Agenda section (Geekathon)

Next, is the "Rules" page for the Geekathon event. It provided participants with detailed guidelines and information about the competition, ensuring every participant had a clear understanding of the event's rules and expectations. This page had no need for database data, so it was simply a static page with multiple sections describing the different rules. As seen in Figure 31, which corresponds to the top of the page, it begins with the name of the event, the date, and a button that when clicked would redirect the user to the page to buy a ticket. Below it, is a recycled component, meaning that it was used in multiple instances to create other sections. That component consists of a title, description, and content, as seen in the bottom half of the Figure 31.

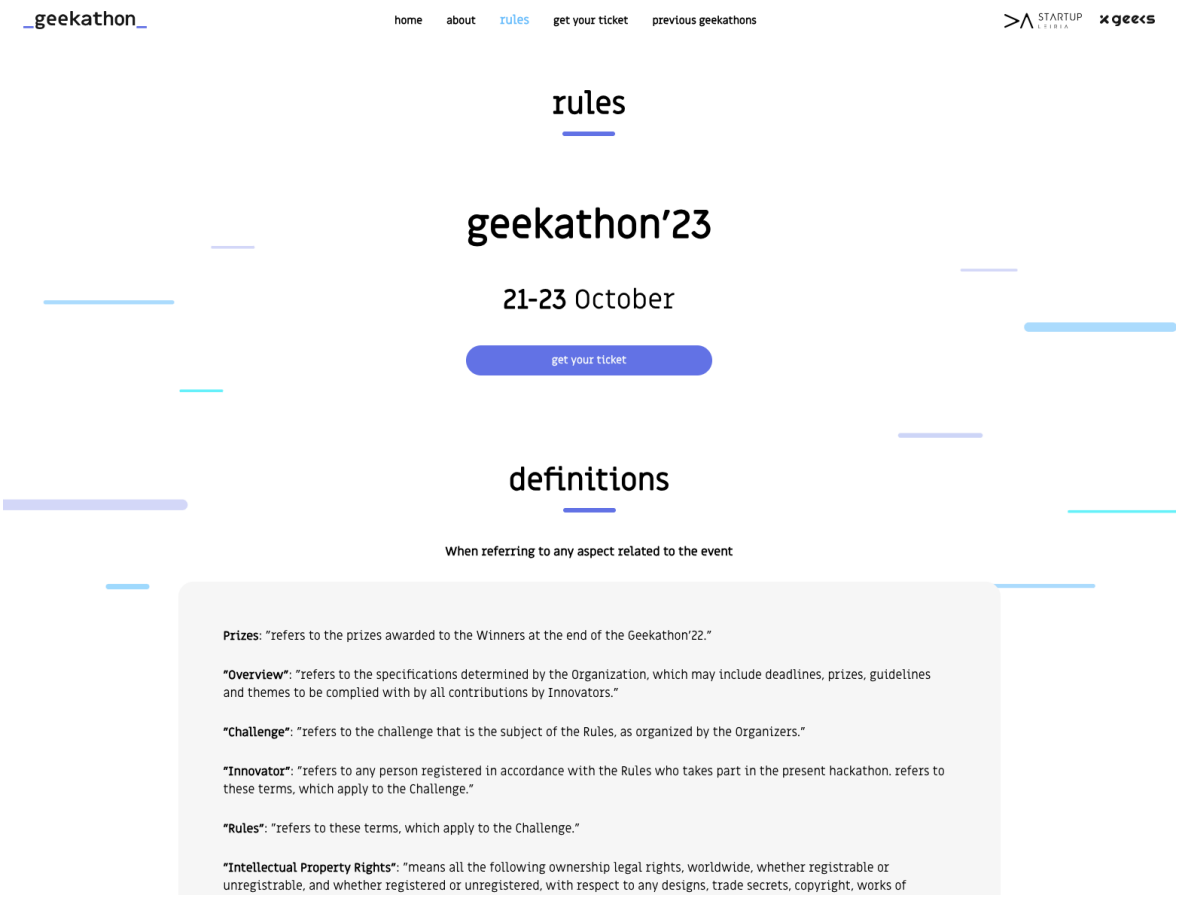

Figure 31 - Rules page (Geekathon)

The last feature I will cover is the "choose your ticket" section for the ticket page. This was one of the main pages of the application, because it was here that the user would have information on what type of tickets exist, as well as their price, information and buy button. Speaking of the buy button, as seen in Figure 32, the button says "register now", it has two different variants, depending if the user was authenticated or not:

- Authenticated: the button said "buy now" and redirected to the payments page;
- Not Authenticated: the button said "register now" and redirected to the registration page.

In terms of design, in case the ticket type was "Super-charged", the card would have a more colorful border, as well as a star icon on the top right corner of the card. The purpose of this was to get the user's attention focused on the most expensive ticket.

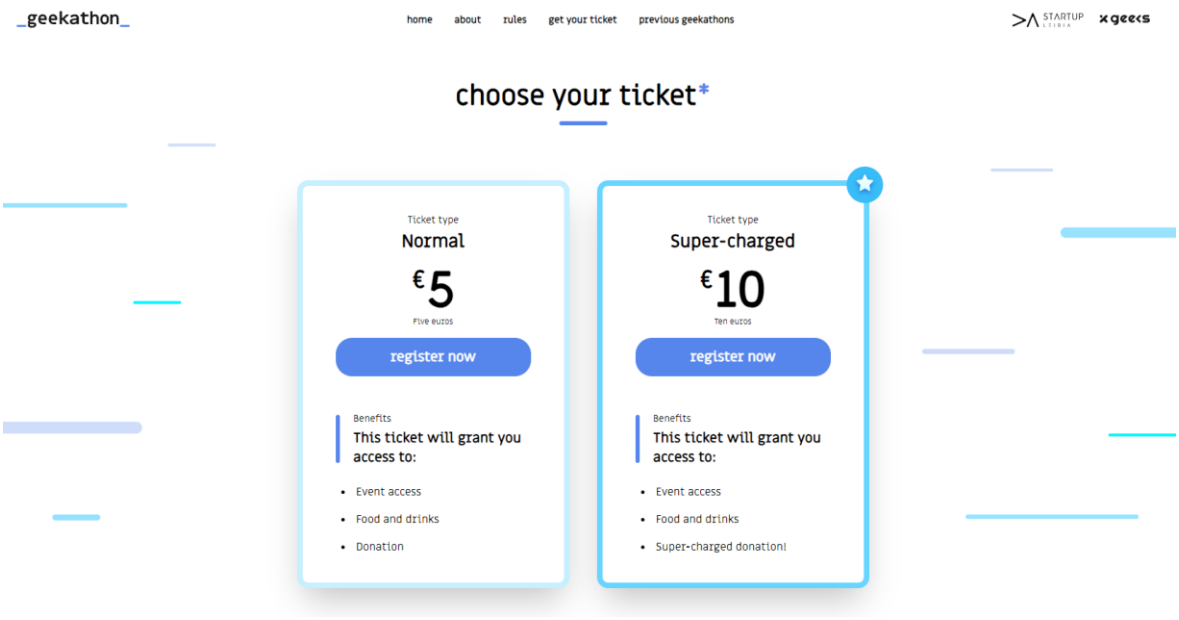

Figure 32 - Choose your ticket page (Geekathon)

For demonstration purposes, below in Figure 33 is shown what the horizontal and small variants did look like, respectively. The information in both variants was much more condensed, displaying only the essential for each use case on the application. There was also a particularity in the horizontal variant, being the "change ticket" hyperlink, responsible for redirecting the user back to the ticket selection page.

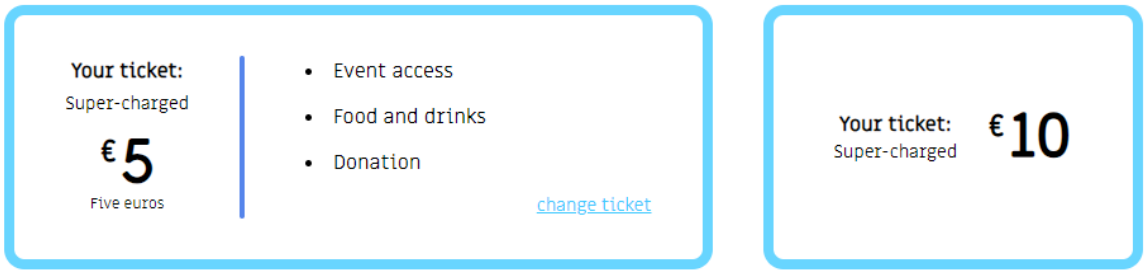

Figure 33 – Ticket Card component horizontal and small variants (Geekathon)

All the data for the cards came from the database, but this time by using the "getServerSideProps" function. Although the information on the cards was not expected to change, the team decided to implement in the project feature flags. The next paragraphs will explain the reason behind using features flags, what they are, and how I was responsible for setting it up. In retrospective, the decision to use "getServerSideProps" instead of

“getStaticProps” will be clear after the next paragraphs (in summary, “getServerSideProps” allowed to check on every request made by the client if the feature should be rendered to the users or not).

The Tech Lead of the project wanted to perform a gradual release of features , so he suggested the use of feature flags. In software development, feature flags (also known as feature toggles or feature switches) are a technique used to enable or disable certain features within an application. They provide a way to control the availability of specific functionalities without the need for deploying new code or creating separate branches.

I volunteered to be responsible for the setup and implementation of a feature flag tool. Given the project’s code was on GitLab, with some research I quickly found out that GitLab itself provides feature flags capabilities, because it natively uses Unleash as a feature toggle service. By looking at the documentation, I was pleased to find a SDK (Software Development Kit) for Next.js, which is the framework we used for the frontend of this project. After that, all I had to do was follow the steps described in the documentation [83]. In order to setup the environment variables, I had to choose between client-side and server-side. I decided to install the SDK using server-side due to two main factors:

1. If using Unleash client-side SDK, it required us to self host Unleash as a service. I spoke with the Dev-Ops team of the project and we decided to avoid it, because they didn’t want another service due to not having enough RAM (Random Access Memory) at the time;
2. If using the server-side option, there was no need to self host the Unleash service. Combined with the fact that I had already developed many pages with SSR, the choice for server-side SDK usage became obvious.

Due to privacy reasons, I was not able to capture an image of the feature flag dashboard. Despite that, I can explain how it worked in practice, by explaining the code used in the “choose your ticket” page (Listing 11). This page made use of the “getServerSideProps” SSR function not only to get the data for the tickets, but also to make use of the “PURCHASE_TICKET” feature flag. The goal for this feature flag was to hide the buy button for the tickets. It began by fetching feature flag definitions using the “getDefinitions” function. The “evaluateFlags” function was then used to evaluate the toggles based on these definitions. The resulting toggles were passed to the “flagsClient” function to obtain a flags object. After that, the code attempted to retrieve pagination data for tickets related to the

current event using the “getTickets” hook. If successful, the function returned an object with the pagination data and a boolean indicating whether the PURCHASE_TICKET flag is enabled or not. In case of an error, the function returned an object with the “notFound” property set to true, which is a Next.js feature that allows the user to be redirected to a “404 not found” page (that page had a custom design, made by another developer).

```
const ChooseYourTicketPage = ({pagination: {data}, isPurchaseEnabled}:
ChooseYourTicketPageProps) => (
  <>
    <Head>
      <title>Get Your Ticket</title>
    </Head>
    <ChooseYourTicket tickets={data} isPurchaseEnabled={isPurchaseEnabled}
/>
    <Perks />
    <TalkToUs />
  </>
);

export const getServerSideProps:
GetServerSideProps<ChooseYourTicketPageProps> = async () => {

  const definitions = await getDefinitions({
    appName: ENV.UNLEASH.APP_NAME,
  });

  const { toggles } = evaluateFlags(definitions);
  const flags = flagsClient(toggles);

  try {
    const pagination = await getTickets(ENV.CURRENT_EVENT);

    return {
      props: { pagination, isPurchaseEnabled:
flags.isEnabled(FEATURE_FLAGS.PURCHASE_TICKET) },
    };
  } catch (error) {
    return { notFound: true };
  }
};

type ChooseYourTicketPageProps = {
  pagination: Pagination<Ticket[]>;
  isPurchaseEnabled: boolean;
};
```

```
export default ChooseYourTicketPage;
```

Listing 11 - Code containing a feature flag (Geekathon)

7.4.Summary

This project consisted of a website for the 2023 Geekathon, a yearly event hosted by xgeeks, similar to a hackathon. In the previous projects I was a Fullstack developer, but for this project I was assigned the role of frontend developer. Another difference is I joined during the early stages of development, instead of the middle or last stages like the other two projects.

The project embraced the Agile methodology, employing "dailies" type meetings three times per week to maintain team cohesion and address challenges. GitLab boards managed tasks, while weekly sync meetings with the product owner ensured alignment and feedback incorporation.

The architecture consisted of Next.js for the frontend and Laravel for the backend. There were some third-party integrations, like Unleash in the frontend for feature management and Stripe in the backend for handling the payments. The application used third-party authentication services such as GitHub, GitLab, Google, Twitter, and LinkedIn. Redis was used as a queueing system, PostgreSQL as a database and Amazon S3 Bucket to store image files.

During the development of this project, I had the initiative to create tasks by myself once I was done completing the tasks assigned to me. I was responsible for creating multiple important features, such as the Navbar, the Footer, home page components, rules page, ticket selection, to name a few. Another functionality I implemented in the application was feature flags, which allowed to turn on/off certain features in the production environment, such as login methods and ability to buy tickets.

8. Conclusion

In retrospect, the internship was a major success because all the goals were accomplished. Not only it was my first time working for a company, but it was also my introduction to the web development world. Although I only had modest knowledge in web development when I started the internship, I was able to quickly learn the fundamentals of the technologies, programming languages and frameworks necessary for my job, such as HTML, CSS, JavaScript, TypeScript, React, Next.js and NestJS. Besides that, I got to put my theoretical knowledge in Agile methodologies into practice by being part of development teams for three different projects, all of which used Agile as their development framework.

In the first two projects, I joined the development teams at the later stages of development, but still managed to contribute to new features, bug fixes and test coverage increase. In the third and last project, I was fortunate to integrate the development team in the early stages of development, so I had first-hand experience on what it was like to build a project from scratch.

In conclusion, I helped come up with ideas, wrote clean and maintainable code, fixed bugs, and provided strong communication in all projects I was part of. This experience was not only personally rewarding but also professionally recognized, as evidenced by the recommendation letter I received from my manager for my exceptional work, which can be seen in Appendix E. It was a remarkable and successful experience, opening the door for me to start my career and find out that frontend development is what I truly aspire to master.

Bibliography

[1] xgeeks, 2023. [Online]. Available: https://xgeeks.io/. [Accessed 28 January 2023].

[2] R. T. Fielding and R. N. Taylor, “Principled Design of the ModernWeb Architecture,” University of California, 2002.

[3] F. Russ, Beginning JavaScript: The Ultimate Guide to Modern JavaScript Development, Apress, 2019.

[4] Wikipedia, “JavaScript,” 4 February 2023. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript. [Accessed 18 February 2023].

[5] Wikipedia, “TypeScript,” 10 February 2023. [Online]. Available: https://en.wikipedia.org/wiki/TypeScript. [Accessed 11 February 2023].

[6] C. Nance, TypeScript Essentials, Packt Publishing Ltd, 2014.

[7] Stack Overflow, “2022 Developer Survey,” May 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies. [Accessed 11 February 2023].

[8] Stack Overflow, “Stack Overflow,” 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018#technology. [Accessed 11 February 2023].

[9] J. Bogner and M. Merkel, “To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022.

[10] GeeksforGeeks, “Difference between TypeScript and JavaScript,” 29 March 2023. [Online]. Available: https://www.geeksforgeeks.org/difference-between-typescript-and-javascript/. [Accessed 6 April 2023].

[11] Stack Overflow, “2022 Developer Survey,” May 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies. [Accessed 11 February 2023].

[12] Stack Overflow, “Developer Survey Results 2018,” 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018#technology. [Accessed 12 April 2023].

[13] P. Rawat and A. N. Mahajan, “ReactJS: A Modern Web Development Framework,” *International Journal of Innovative Science and Research Technology*, p. 5, 2020.

[14] A. Banks and E. Porcello, Learning React: functional web development with React and Redux, O'Reilly Media, Inc., 2017.

[15] D. Bugl, Learn React Hooks: Build and refactor modern React. js applications using Hooks, Packt Publishing Ltd, 2019.

[16] Testing Library, “Testing Library,” 2023. [Online]. Available: https://testing-library.com/docs/react-testing-library/intro. [Accessed 19 April 2023].

[17] JestJS, “Jest - Delightful JavaScript Testing,” 2023. [Online]. Available: https://jestjs.io/. [Accessed 19 April 2023].

[18] Next.js, “What is Next.js,” 2023. [Online]. Available: https://nextjs.org/learn/foundations/about-nextjs/what-is-nextjs. [Accessed 12 April 2023].

[19] H. Jartarghar, G. Salanke, A. A.R, S. G.S and S. Dalali, “React Apps with Server-Side Rendering: Next.js,” *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, pp. 25-29, 2022.

[20] Next.js, “getServerSideProps,” 2023. [Online]. Available: https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props. [Accessed 13 April 2023].

[21] Next.js, “getStaticProps,” 2023. [Online]. Available: https://nextjs.org/docs/basic-features/data-fetching/get-static-props. [Accessed 13 April 2023].

[22] Next.js, “Incremental Static Regeneration,” 2023. [Online]. Available: https://nextjs.org/docs/basic-features/data-fetching/incremental-static-regeneration. [Accessed 13 April 2023].

[23] Next.js, “getStaticPaths,” 2023. [Online]. Available: https://nextjs.org/docs/basic-features/data-fetching/get-static-paths. [Accessed 13 April 2023].

[24] Next.js, “Client-side data fetching,” 2023. [Online]. Available: https://nextjs.org/docs/basic-features/data-fetching/client-side. [Accessed 13 April 2023].

[25] Next.js, “Routing,” 2023. [Online]. Available: https://nextjs.org/docs/routing/introduction. [Accessed 14 April 2023].

[26] Next.js, “Dynamic Routes,” 2023. [Online]. Available: https://nextjs.org/docs/routing/dynamic-routes. [Accessed 16 April 2023].

[27] Next.js, “API Routes,” 2023. [Online]. Available: https://nextjs.org/docs/api-routes/introduction. [Accessed 16 April 2023].

[28] M. Lazuardy and D. Anggraini, “Modern Front End Web Architectures with React.Js and Next.Js,” *International Research Journal of Advanced Engineering and Science,* pp. 132-141, 2022.

[29] Tanstack, “Overview,” [Online]. Available: https://tanstack.com/query/v4/docs/react/overview. [Accessed 11 March 2023].

[30] Redux, “Redux Essentials, Part 1: Redux Overview and Concepts,” [Online]. Available: https://redux.js.org/tutorials/essentials/part-1-overview-concepts. [Accessed 11 March 2023].

[31] E. Meyer, CSS: The Definitive Guide, O'Reilly Media, Inc., 2006.

[32] Wikipedia, “Tailwind CSS,” 7 March 2023. [Online]. Available: https://en.wikipedia.org/wiki/Tailwind_CSS. [Accessed 18 March 2023].

[33] Tailwind Labs, “Beautiful UI components, crafted with Tailwind CSS,” [Online]. Available: https://tailwindui.com/components?ref=sidebar. [Accessed 18 March 2023].

[34] Radix, “Introduction,” [Online]. Available: https://www.radix-ui.com/docs/primitives/overview/introduction. [Accessed 18 March 2023].

[35] Stitches, “Introduction,” [Online]. Available: https://stitches.dev/docs/introduction. [Accessed 18 March 2023].

[36] Mozilla, “HTML: HyperText Markup Language,” 24 February 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML. [Accessed 11 March 2023].

[37] Typescriptlang, “JSX,” [Online]. Available: https://www.typescriptlang.org/docs/handbook/jsx.html. [Accessed 11 March 2023].

[38] NestJS, “Introduction,” 2023. [Online]. Available: https://docs.nestjs.com/. [Accessed 11 April 2023].

[39] NestJS, “CLI Overview,” 2023. [Online]. Available: https://docs.nestjs.com/cli/overview. [Accessed 11 April 2023].

[40] NestJS, “Custom route decorators,” 2023. [Online]. Available: https://docs.nestjs.com/custom-decorators. [Accessed 11 April 2023].

[41] NestJS, “Controllers,” 2023. [Online]. Available: https://docs.nestjs.com/controllers. [Accessed 11 April 2023].

[42] NestJS, “Providers,” 2023. [Online]. Available: https://docs.nestjs.com/providers. [Accessed 11 April 2023].

[43] NestJS, “Modules,” 2023. [Online]. Available: https://docs.nestjs.com/modules. [Accessed 11 April 2023].

[44] NestJS, “Guards,” 2023. [Online]. Available: https://docs.nestjs.com/guards. [Accessed 11 April 2023].

[45] NestJS, “Testing,” 2023. [Online]. Available: https://docs.nestjs.com/fundamentals/testing. [Accessed 19 April 2023].

[46] A. D. Pham, “Developing back-end of a web application with NestJS framework: Case: Integrify Oy’s student management system.,” Lab University of Applied Sciences LTD, 2020.

[47] NestJS, “SQL (TypeORM),” 2023. [Online]. Available: https://docs.nestjs.com/recipes/sql-typeorm. [Accessed 7 April 2023].

[48] TypeORM, “Getting Started,” 2023. [Online]. Available: https://typeorm.io/. [Accessed 7 April 2023].

[49] TypeORM, “Entities,” 2023. [Online]. Available: https://typeorm.io/entities. [Accessed 7 April 2023].

[50] TypeORM, “Relations,” 2023. [Online]. Available: https://typeorm.io/relations. [Accessed 7 April 2023].

[51] K. Banker, P. Bakkum, S. Verch, D. Garret and T. Hawkins, MongoDB in Action, Second Edition: Covers MongoDB version 3.0, Manning Publications, 2016.

[52] A. Kukic and S. Vlaeva, “MongoDB & Mongoose: Compatibility and Comparison,” 25 November 2022. [Online]. Available:

https://www.mongodb.com/developer/languages/javascript/mongoose-versus-nodejs-driver/. [Accessed 11 March 2023].

[53] Mongoose, "Guides," [Online]. Available: https://mongoosejs.com/docs/guides.html. [Accessed 11 March 2023].

[54] A. Chauhan, "A Review on Various Aspects of MongoDb Databases," *International Journal of Engineering Research & Technology (IJERT),* pp. 90-92, 2019.

[55] J. Worsley and J. Drake, Pratical PostgreSQL, O'Reilly Media, Inc., 2002.

[56] S. Juba, A. Vannahme and A. Volkov, Learning PostgreSQL, Packt Publishing Ltd., 2015.

[57] Redis, "Introduction to Redis," [Online]. Available: https://redis.io/docs/about/. [Accessed 4 March 2023].

[58] J. Nelson, Mastering Redis, Packt Publishing Ltd, 2016.

[59] D. Eddelbuettel, *A Brief Introduction to Redis,* University of Illinois Urbana-Champaign, 2022.

[60] Mongodb, "MongoDB vs Redis Comparison," [Online]. Available: https://www.mongodb.com/compare/mongodb-vs-redis. [Accessed 11 March 2023].

[61] S. Al-Saqqa, S. Sawalha and H. AbdelNabi, "Agile Software Development: Methodologies and Trends," *International Journal of Interactive Mobile Technologies,* p. 25, 2020.

[62] M. Fowler and J. Highsmith, "The Agile Manifesto," Agile Alliance, Utah, 2001.

[63] I. Perera and S. Arachchi, "Continuous integration and continuous delivery pipeline automation for agile software project management," Engineering Research Conference, Moratuwa, 2018.

[64] A. Srivastava, S. Bhardwaj and S. Saraswat, “SCRUM Model for Agile Methodology,” in *International Conference on Computing, Communication and Automation*, 2017.

[65] R. Wakode, L. Raut and P. Talmale, “Overview on kanban methodology and its implementation.,” *IJSRD-International Journal for Scientific Research & Development*, pp. 2321-0613, 2015.

[66] M. Ahmad, J. Markkula and M. Oivo, “Kanban in software development: A systematic literature review,” in *39th Euromicro conference on software engineering and advanced applications*, 2013.

[67] Agile Alliance, “Scrumban,” 2023. [Online]. Available: https://www.agilealliance.org/scrumban/. [Accessed 3 May 2023].

[68] L. Williams, “Pair Programming,” Encyclopedia of software engineering 2, North Carolina, 2010.

[69] A. Begel and N. Nagappan, “Pair Programming: What’s in it for Me?,” Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008.

[70] xgeekshq, “SPLIT,” 14 June 2023. [Online]. Available: https://github.com/xgeekshq/split. [Accessed 3 May 2023].

[71] EasyRetro, “Improve with Fun Sprint Retrospectives,” 2023. [Online]. Available: https://easyretro.io/. [Accessed 3 May 2023].

[72] TeamRetro, “Run Online Retrospectives with TeamRetro,” 2023. [Online]. Available: https://www.teamretro.com/. [Accessed 3 May 2023].

[73] Reetro, “Free Online Retrospectives,” 2023. [Online]. Available: https://reetro.io/. [Accessed 3 May 2023].

[74] Okta, “What is OAuth 2.0?,” 2023. [Online]. Available: https://auth0.com/intro-to-iam/what-is-oauth-2. [Accessed 27 July 2023].

[75] React, “Suspense,” [Online]. Available: https://react.dev/reference/react/Suspense. [Accessed 28 May 2023].

[76] TanStack, “Infinite Queries,” [Online]. Available: https://tanstack.com/query/v4/docs/react/guides/infinite-queries. [Accessed 4 June 2023].

[77] react-i18next, “useTranslation (hook),” 2023. [Online]. Available: https://react.i18next.com/latest/usetranslation-hook. [Accessed 22 April 2023].

[78] npmjs, “yup,” 2023. [Online]. Available: https://www.npmjs.com/package/yup. [Accessed 23 April 2023].

[79] NestJS, “Testing,” [Online]. Available: https://docs.nestjs.com/fundamentals/testing. [Accessed 31 March 2023].

[80] Laravel, “The PHP Framework for Web Artisans,” 2023. [Online]. Available: https://laravel.com/. [Accessed 27 July 2023].

[81] Unleash, “Code freely. Release confidently. Then scale it,” 2023. [Online]. Available: https://www.getunleash.io/. [Accessed 18 July 2023].

[82] Sinch, “Mailgun,” 2023. [Online]. Available: https://www.mailgun.com/. [Accessed 27 July 2023].

[83] Unleash, “Next.js,” 2023. [Online]. Available: https://docs.getunleash.io/reference/sdks/next-js. [Accessed 18 July 2023].

[84] Beekai, “useForm,” [Online]. Available: https://react-hook-form.com/api/useform/. [Accessed 28 March 2023].

[85] npmjs, “classnames,” 2023. [Online]. Available: https://www.npmjs.com/package/classnames. [Accessed 1 May 2023].

[86] B. Rad, H. Bhatti and M. Ahmadi, “An Introduction to Docker and Analysis of its Performance,” *International Journal of Computer Science and Network Security,* 2017.

[87] A. Rashid and A. Chaturvedi, “Cloud Computing Characteristics and Services: A Brief Review,” *International Journal of Computer Sciences and Engineering*, pp. 421-426, 2019.

[88] M. Collier and R. Shahan, Microsoft Azure Essentials - Fundamentals of Azure, Microsoft Press, 2015.

[89] M. Asaswat and R. Tripathi, “Cloud Computing: Comparison and Analysis of Cloud Service Providers—AWS, Microsoft and Google,” in *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, Moradabad, 2020.

[90] Microsoft, “Azure for AWS professionals,” 12 December 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/aws-professional/. [Accessed 25 March 2023].

[91] Microsoft, “Azure for Google Cloud Professionals,” 30 November 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/gcp-professional/. [Accessed 25 March 2023].

Appendices

Appendix A

The IntegrationCard component is where all the major functionality of this page is (Listing 12). It starts by defining the default value for the handle in the input, which can either be empty or contain the handle of the current user for that given Integration (both scenarios can be seen in Figure 23). Then there is the “useState” hook for tracking whether the form has been submitted, which is important because this will affect the state of the CSS of the input element, as well as the state of the submit button (disabled after submission or with errors, otherwise enabled). Throughout the application, every form integrates the custom hook “useForm” provided by the React Hook Form library [84], as it is a great tool to manage form state in React or Next.js applications. The hook is being stored into a variable and takes an object with three properties: “resolver”, “mode” and “defaultValues”. The “resolver” property is a function used for form validation, which in this case uses the yup schema builder library [78] that takes in a yup object with validation rules. Then the “mode” property specifies to trigger form validation every time there is a change to the input, and the “defaultValues” property specifies the initial value of the handle. Several properties are extracted from the “useFormState” variable, starting by the “formState” property, which provides access to the state of the form, including whether the form is valid based on the defined validation rules (“isValid” property), and whether the form values have been changed (“isDirty” property). The “handleSubmit” function is used as the form's “onSubmit” callback function, and it handles the submission of the form data when the submit button is clicked. At last, the “reset” function allows to specify which values correspond to the initial data. These properties make it easy to create forms with validation logic and dynamic behaviour in a React or Next.js app.

The “useEffect” hook in this case allows for the component to re-render when it receives a handle as a prop, or in other words, in case the user already had submitted the handle for that given Integration it will show that handle already written in the input field, just as a quality-of-life feature. Finally, before the return statement to render the UI elements, there is the “useMutation” hook, which is called in the “submitForm” function, which subsequentially is called when the user submits a valid handle. The “useMutation” hook receives two properties, the handle, and the integration name, and makes an API call to the backend to create/update the handle of the user on that specific integration. In case of

success, the new default value for the input will be the new handle and the user will be notified with a pop-up notification of success, otherwise the pop-up will tell the user what went wrong in the request.

```
const IntegrationCard = ({
  image,
  title,
  provider,
  formSchema,
  description,
  placeholder,
  handle,
  'data-testid': dataTestId = 'integration-card',
  ...props
}: IntegrationCardProps) => {
  const defaultValues: DefaultValues<IntegrationFormData> = {
    handle,
  };
  const [formSubmitted, setFormSubmitted] = useState(false);

  const useFormState = useForm<IntegrationFormData>({
    resolver: yupResolver(formSchema),
    mode: 'onChange',
    defaultValues,
  });

  const {
    formState: { isValid, isDirty },
    handleSubmit,
    reset,
  } = useFormState;

  useEffect(() => {
    reset(defaultValues);
  }, [handle]);

  const inputBorder = inputCSS(formSubmitted, isDirty, handle);

  const { mutate: addIntegration } = useMutation<
    IntegrationResponse,
    ErrorWithCause,
    IntegrationRequest
  >(
    ['addIntegration'],
    (integrationData: IntegrationRequest) =>
      integrationService.addIntegration(
        integrationData.provider,
        integrationData.externalUserHandle,
```

```
      ),
    {
      onSuccess: (response) => {
        setFormSubmitted(true);
        const newHandle = {
          handle: response.externalUserHandle,
        };
        reset(newHandle);

        toast.success(`${title} handle '${response.externalUserHandle}'
successfully submited`);
      },
      onError: (error) => {
        toast.error(treatErrorMessage(error.cause.message));
      },
    },
  );

  const submitForm = (data: IntegrationFormData) => {
    addIntegration({ provider, externalUserHandle: data.handle });
  };

  return (
    <div className="card bg-white shadow-sm rounded-xl border" data-
testid={dataTestId} {...props}>
      <Form useFormState={useFormState}
onSubmit={handleSubmit(submitForm)}>
        <div className="card-body px-5 py-6">
          <Image src={image} alt={title} height="48" width="48" />

          <h2 className="card-title text-2xl font-semibold" data-
testid="title">
            {title}
          </h2>

          <Text
            content={description}
            customStyle="text-base break-words text-neutral-500"
            data-testid="description"
          />

          <div className="flex mt-4 gap-4">
            <Input
              id="handle"
              variant={inputBorder}
              fieldName="handle"
              placeholder={placeholder}
              data-testid="input-handle"
            />
```

```
            <button
              className="btn bg-neutral-500 border-none w-14 rounded-md
hover:bg-emerald-500"
              type="submit"
              disabled={!isValid || !isDirty}
              data-testid="submit-handle"
            >
              <TbCheck size={22} />
            </button>
          </div>
        </div>
      </Form>
    </div>
  );
};

export type IntegrationCardProps = {
  image: string;
  title: string;
  provider: Provider;
  formSchema: yup.ObjectSchema<any, Record<string, any>, any>;
  description: string;
  placeholder: string;
  handle: string;
  'data-testid'?: string;
};

export default IntegrationCard;
```

Listing 12 - IntegrationCard (xAchievements)

The second part of the code renders a card with a form that contains an image, a title, a description, an input field, and a submit button. The component receives props including a data-testid, an image source URL (to show the GitHub/GitLab logo), a title, a description, a placeholder text for the input field, and other Form and Input props. The Form component is used to manage the form state, and the “onSubmit” handler is used to submit the form data when the submit button is clicked. The “handleSubmit” function is called when the form is submitted. The Image and Text components are used to render the image and description, respectively. The Input component is used to render the input field, and the submit button contains a checkmark icon.

Appendix B

The first part of the IntegrationCard test file starts by creating a render function for that component (Listing 13). In this case, the component will be rendered by default with data corresponding to the GitHub provider. The “renderWithProviders” function is a utility function created in order to return a component rendered with a mocked query client and router. The first test, like every other component in the application, starts by checking if the component renders correctly. After that, the next iteration tests the scenario where a “User already has a handle and can’t submit without changing it”. In that case, the expected behaviour is for the submit button to be disabled, because the form is already filled with the current handle of the user, so the button will only be enabled if a change is made. In order to test that, the component is rendered with an existing handle and the “getByTestId” function is extracted from the render, which allows to select elements in a component by their “data-testid” attribute. With that said, it is expected for the input element to have the same handle that was inserted on render. After that, the submit button is stored into a variable and it is expected to the disabled because the user already has a handle.

```
import { getGithubFormSchema, getGitlabFormSchema } from
'@/constants/integrations';
import { ProviderEnum } from '@/types/settings';
import { renderWithProviders } from '@/utils/testing';
import { waitFor } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { useTranslation } from 'next-i18next';
import IntegrationCard, { IntegrationCardProps } from './IntegrationCard';

const render = (props: Partial<IntegrationCardProps> = {}) => {
  const { t } = useTranslation('settings');

  return renderWithProviders(
    <IntegrationCard
      image="/github-logo.png"
      title="GitHub"
      provider={ProviderEnum.Github}
      formSchema={getGithubFormSchema(t)}
      description="Description"
      placeholder="Write Something"
      handle=""
      {...props}
    />,
  );
};
```

```
describe('IntegrationCard', () => {
  it('Renders successfully', () => {
    expect.assertions(1);

    const { getByTestId } = render();

    expect(getByTestId('integration-card')).toBeInTheDocument();
  });

  it('User already has a handle and cant submit without changing it', ()
=> {
    const { getByTestId } = render({ handle: 'MrTest' });

    expect(getByTestId('input-handle')).toHaveValue('MrTest');

    const submitHandleButton = getByTestId('submit-handle') as
HTMLButtonElement;

    expect(submitHandleButton.disabled).toBe(true);
  });
```

Listing 13 - IntegrationCard component test file 1/2 (xAchievements)

Moving on to the next test iterations, in Listing 14, the “Handle is changed and the submit button becomes enabled” test assumes the user had previously submitted a handle and tries to change it. In order to simulate a user event, the function “userEvent” is used. The purpose is to type into the input element the new handle and it is good practice to wait for user events to finish because they are asynchronous, meaning that if not await, the code can still move to new lines and cause errors because the user event action is not fully complete. With that done, the iteration finishes by expecting the button to become enabled, because the handle is now different so the user should be able to click the submit button in case, he wants to change the handle to a new one

Finally, the last two iterations test the scenarios where the user inputs an invalid handle for both providers (GitLab and GitHub). Since both tests are similar, I will just explain one, the GitLab provider. Because the initial render function in Listing 14 uses the GitHub data as props by default, in order to test the GitLab provider I needed to overwrite the props in order to fit the GitLab provider context. I researched both GitHub and GitLab usernames validation, and implemented them in the form validation, so in this test it is being simulated that the user is typing a handle with only one character, which GitLab does not allow, so it is expected that the submit button will be disabled.

```
it('Handle is changed and the submit button becomes enabled', async () =>
{
    const { getByTestId } = render();

    const newHandle = 'newHandle';
    const inputElement = getByTestId('input-handle') as HTMLInputElement;
    userEvent.type(inputElement, newHandle);
    await waitFor(() => expect(inputElement.value).toBe(newHandle));

    const submitHandleButton = getByTestId('submit-handle');

    expect(submitHandleButton).toBeEnabled();
  });

  it('Invalid GitHub handle', async () => {
    const { getByTestId } = render();

    const githubHandle = 'git--';
    const inputElement = getByTestId('input-handle') as HTMLInputElement;
    userEvent.type(inputElement, githubHandle);
    await waitFor(() => expect(inputElement.value).toBe(githubHandle));

    const submitHandleButton = getByTestId('submit-handle');

    expect(submitHandleButton).toBeDisabled();
  });

  it('Invalid GitLab handle', async () => {
    const { t } = useTranslation('settings');

    const { getByTestId } = render({
      title: 'GitLab',
      provider: ProviderEnum.Gitlab,
      formSchema: getGitlabFormSchema(t),
    });

    const gitlabHandle = 'n';
    const inputElement = getByTestId('input-handle') as HTMLInputElement;
    userEvent.type(inputElement, gitlabHandle);
    await waitFor(() => expect(inputElement.value).toBe(gitlabHandle));

    const submitHandleButton = getByTestId('submit-handle');

    expect(submitHandleButton).toBeDisabled();

  });
```

Listing 14 - IntegrationCard component test file 2/2 (xAchievements)

Appendix C

The Button component is highly configurable and allows for customization of the button's size, variant, style, and other props, as seen in the code (Listing 15). The Button component is built using the “classNames” library [85], which helps to conditionally concatenate CSS classes based on the props passed to the component. The “classNames” function is called with several arguments, which are the different CSS classes for the button. These classes are defined in the BUTTON_CLASSES object, which provides a set of pre-defined styles for the button component. The props accepted by the Button component are defined by the “ButtonProps” interface. The interface extends the intrinsic button element props, allowing the Button component to accept all standard button props such as “onClick”, disabled, etc. Additionally, the interface defines a set of custom props, such as data-testid, size, variant, style, and icon. These props provide additional customization options for the button component.

```
const BUTTON_CLASSES = {
  base: 'btn rounded gap-2 normal-case',
  size: {
    extraSmall: 'btn-xs',
    small: 'btn-sm',
    medium: 'btn-md',
  },
  variant: {
    primary: 'btn-primary',
    secondary: 'btn-secondary text-white hover:!text-white
[&>*]:hover:text-white',
    error: 'btn-error text-white hover:!text-white',
    success: 'btn-success text-white hover:!text-white',
    danger: 'bg-red-500 hover:bg-red-700 text-white border-transparent
hover:border-transparent',
  },
  style: {
    outline: 'btn-outline border',
  },
};

const Button = ({
  type = 'button',
  'data-testid': dataTestId = 'button',
  children,
  size = 'small',
  variant,
  style,
```

```
  className,
  icon,
  ...props
}: ButtonProps) => (
  <button
    type={type}
    data-testid={dataTestId}
    {...props}
    className={classNames(
      BUTTON_CLASSES.base,
      BUTTON_CLASSES.size[size],
      variant ? BUTTON_CLASSES.variant[variant] : '',
      style ? BUTTON_CLASSES.style[style] : '',
      className,
    )}
  >
    {children}
    {icon}
  </button>
);

export type ButtonProps = Omit<JSX.IntrinsicElements['button'], 'style'> &
{
  'data-testid'?: string;
  size?: 'extraSmall' | 'small' | 'medium';
  variant?: 'primary' | 'secondary' | 'error' | 'success' | 'danger';
  style?: 'outline';
  icon?: ReactElement;
};


export default Button;
```

Listing 15 - Button Primitive component code (xAchievements)

Appendix D

When I joined the project, there was already a page with the list of rewards, but it was not complete. I fixed some bugs in this page, such as showing the correct rewards when filtered by name, as well as some new features, like the ability to filter rewards by their status (Pending, Delivered or All). Figure 34 represents the Rewards page with the filter to show all rewards.

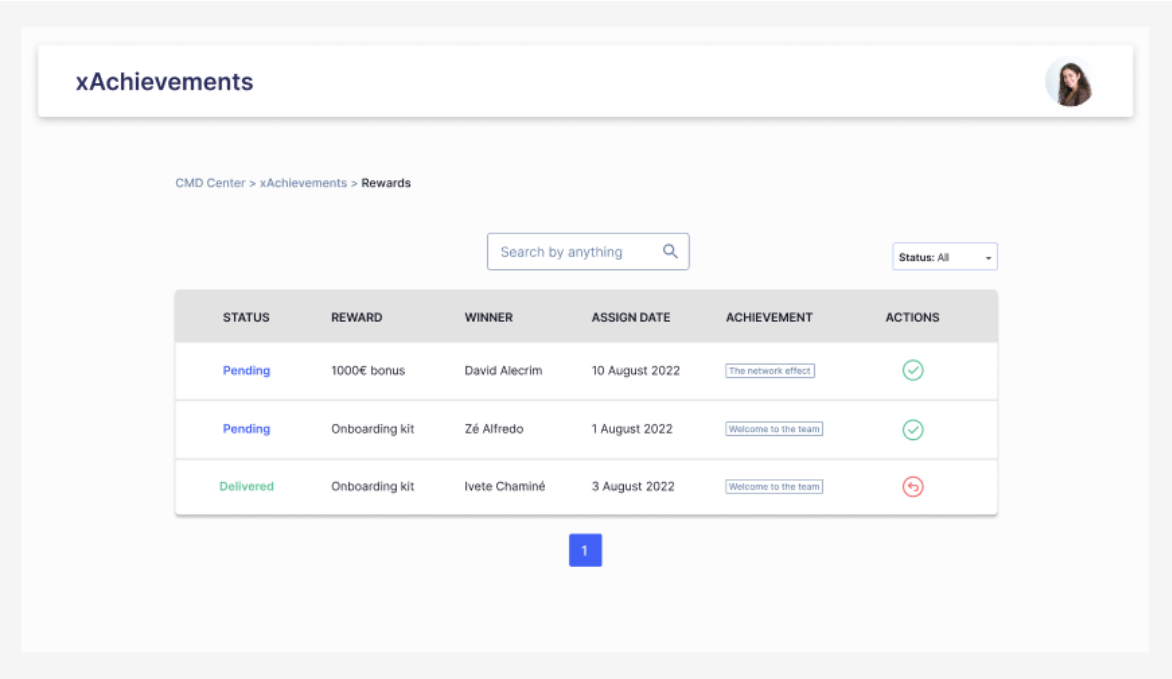

Figure 34 - Rewards page (xAchievements)

Another feature I developed in this page was to select one or more rewards and in case they were Pending, mark them as delivered, updating their status on the database. If the rewards had the Delivered status, the same logic would apply, in order to revert their status back to Pending. The Figma design for this feature can be seen below in Figure 35 and Figure 36.

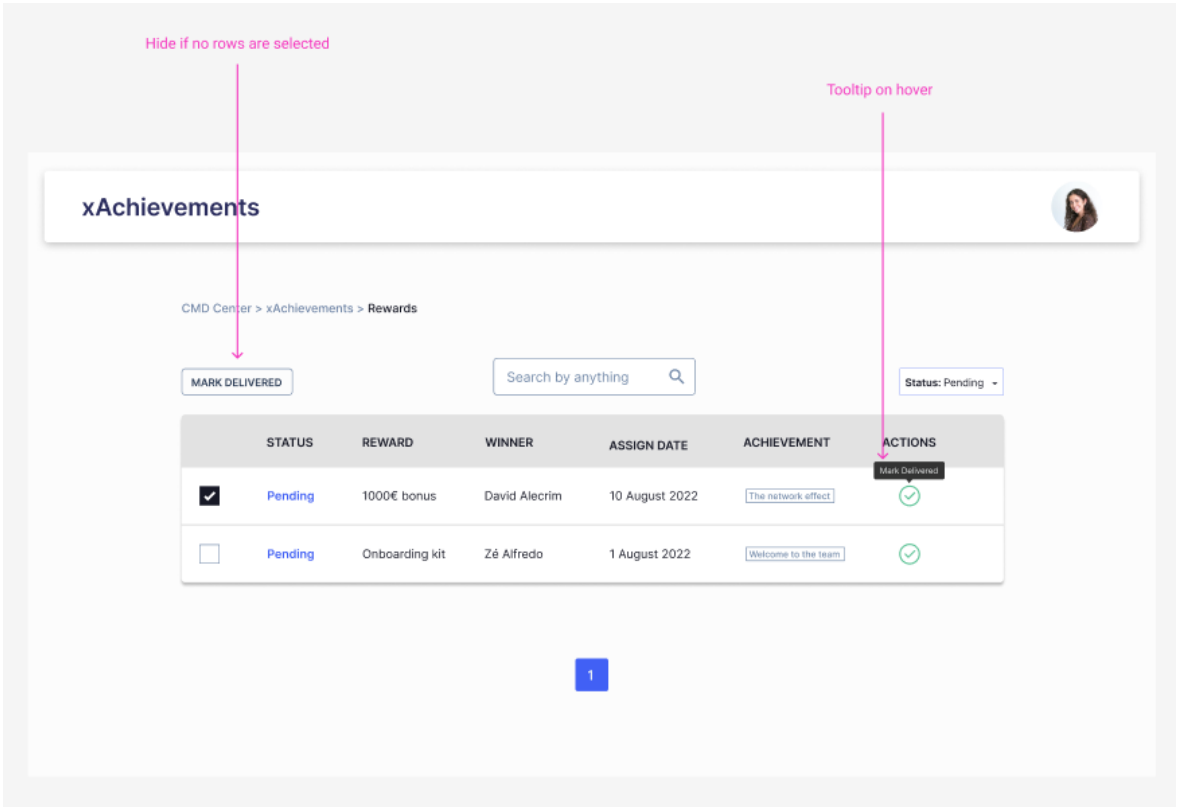

Figure 35 - Mark rewards as Delivered (xAchievements)

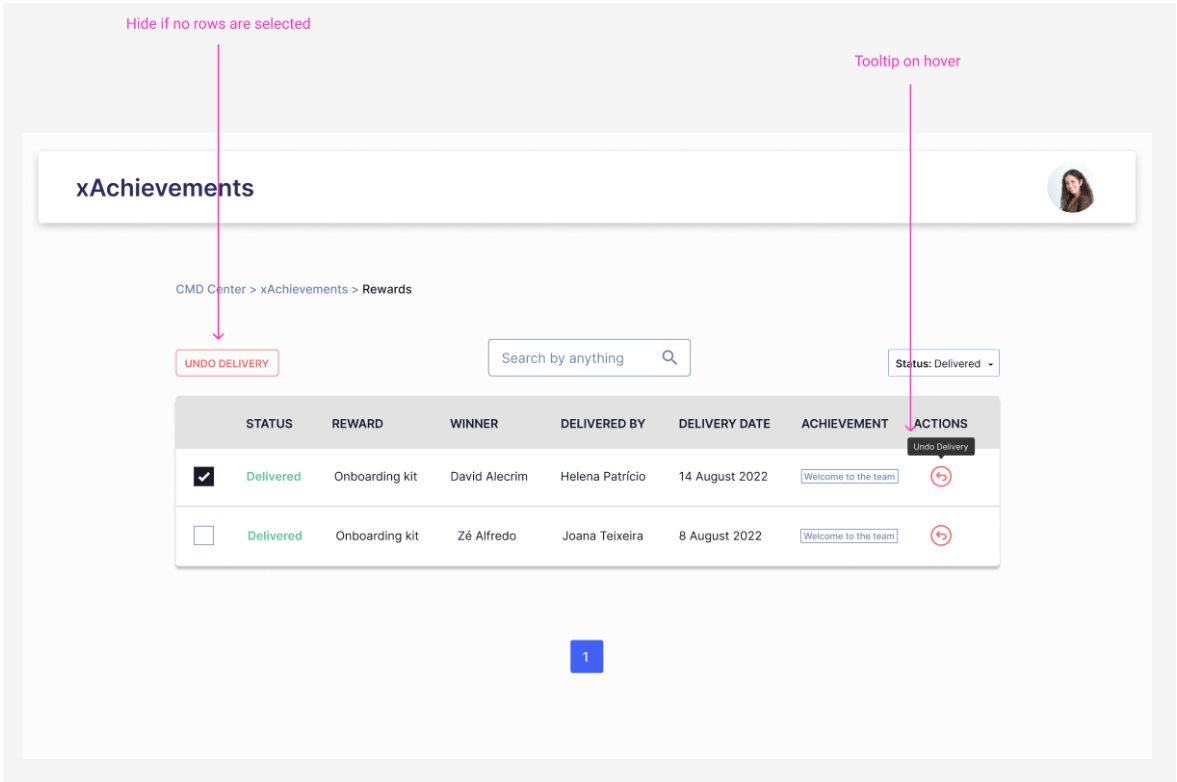

Figure 36 - Undo Delivered rewards (xAchievements)

When looking at the details of an achievement, it was possible to see the list of users who completed that achievement. In addition to that, I developed the functionality that allowed to click on the row of a user (Figure 37), resulting in the application redirecting to the profile of that user.

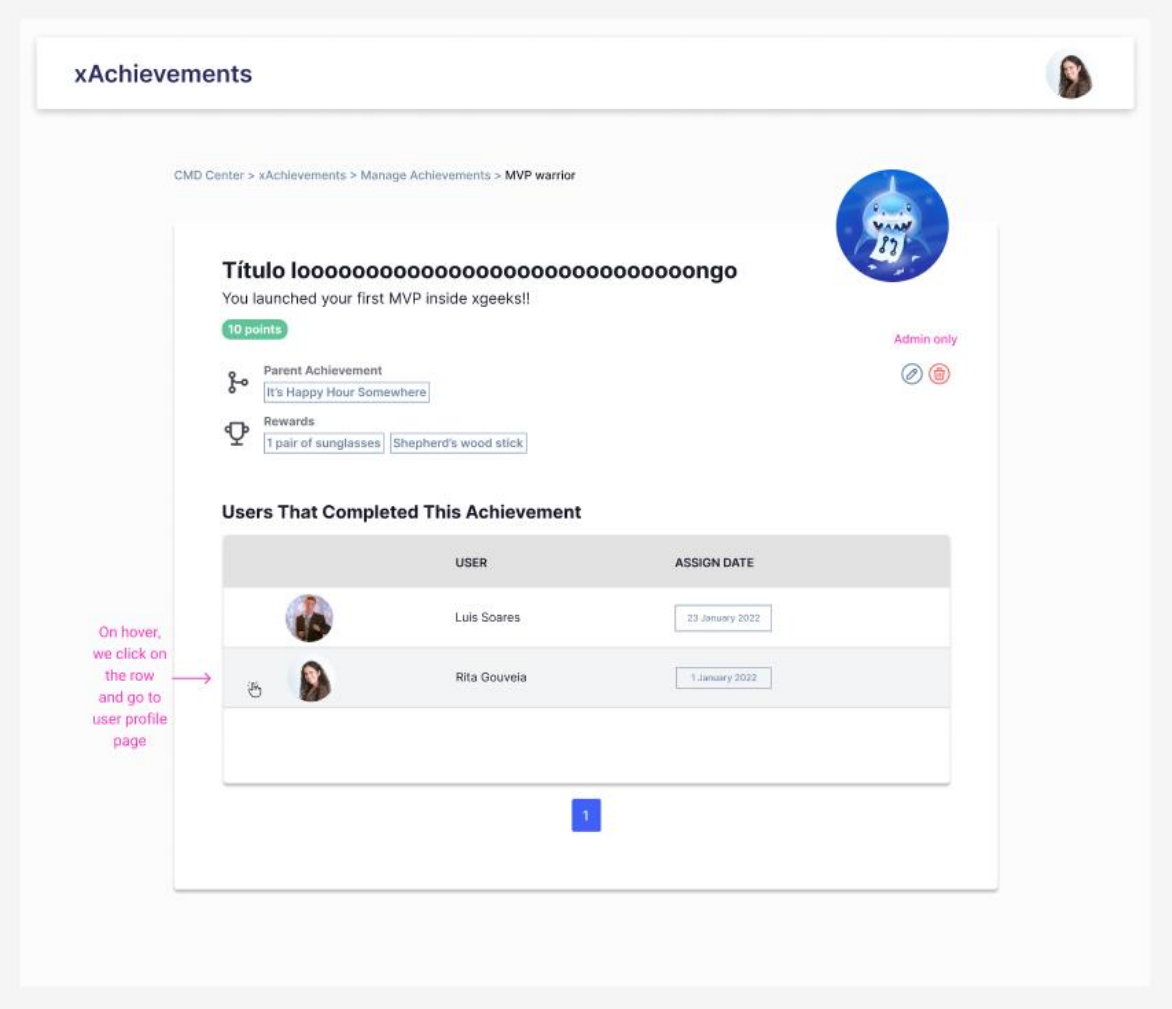

Figure 37 - Achievement details page (xAchievements)

Appendix E

As recognition for my excellent work during the internship, I was given a recommendation letter (signed by my manager at xgeeks), shown in Figure 38.

July 7, 2023

To Whom This May Concern,

My name is Gonçalo Dias, Engineering Manager at xgeeks.
I had the pleasure of working closely with Guilherme during his tenure at xgeeks, and I am confident in his ability to make a significant impact in any engineering team he joins.

During his time at xgeeks, Guilherme made an impactful contribution to our internal projects. His ability to quickly grasp new concepts and technologies allowed him to become productive from day one. He consistently demonstrated good organizational skills, ensuring that his work was always completed on time and matching our engineering standards.

Guilherme as a complete grasp of modern full-stack web technologies. Most notably, he has acquired a deep expertise in frontend development using TypeScript and Next.js. Guilherme consistently brought innovative ideas to the table, enhancing the overall quality of our products.

Moreover, Guilherme as shown strong communication skills, both with their peers and stakeholders. He actively seeks input and feedback, ensuring a collaborative and productive work environment.

I believe that Guilherme has tremendous potential for growth. He has consistently demonstrated a strong desire to expand his skill set and tackle new challenges. With his curiosity and willingness to learn, I have no doubt that Guilherme will continue to excel in his professional development.

In conclusion, I wholeheartedly recommend Guilherme Santo for any engineering position within your organization. I am confident that he'll make significant contributions to your team, and his presence will be a valuable addition to your organization.

Thank you for considering my recommendation.

Sincerely,

Engineering Manager
xgeeks

Figure 38 - Recommendation Letter